An Analysis-Driven Rapid Design Process for Cyber-Physical Systems

By

Zsolt Lattmann

Dissertation
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY
in

Electrical Engineering

August, 2016

Nashville, Tennessee

Approved:

Gábor Karsai, Ph.D.

Theodore Bapty, Ph.D.

Gautam Biswas, Ph.D.

Xenofon Koutsoukos, Ph.D.

Sandeep Neema, Ph.D.

János Sztipanovits, Ph.D.

ACKNOWLEDGMENTS

First and foremost I wish to thank my advisor, Prof. Gábor Karsai, for his scientific advice and knowledge and many insightful discussions and suggestions. He helped me to formulate the thesis topic and guided me over almost four years of development. His questions were always the most difficult ones, and they guided me towards research challenges, answers, and solutions. He helped me learn how to clearly and precisely communicate my ideas. He is the person who reads every single word on a page, even the fine print, and asks questions about every figure and diagram. His thorough work advocates excellence and the highest quality work based on all available information and time constraints. He clearly had a positive influence on my personality and professional career. He taught me how to effectively manage my time and be more productive; as a result, over the years at Vanderbilt University I have rarely missed deadlines, or even requested a deadline extension. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. I am also thankful for the excellent example he has provided as a successful mentor, scientist, and professor.

I would like to thank James Klingler for helping me with his proofreading and continuous feedback on technical content and suggesting how to rephrase and reorganize certain portions of this thesis to make it easier to read and understand. Even when I was losing my motivation he encouraged me to improve the quality this document. As I learn more and more about writing, I realize how much I do not know. I truly believe that James has just started me on a journey to improving my writing skills even further. Thank you, James.

My thesis committee has given me exceptional input through all these years. Thank you to Prof. János Sztipanovits, Prof. Xenofon Koutsoukos, Prof. Gautam Biswas, Prof. Sandeep Neema, and Prof. Ted Bapty. I would like to thank all of my committee members for their valuable feedback, which has greatly improved the quality and clarity of this thesis and helped me define the scope of this research. Prof. Sandeep Neema and Prof. Ted Bapty provided me with the opportunity to be a contributor to the Defense Advanced Research Projects Agency (DARPA) Adaptive Vehicle Make (AVM) program and several projects including the META project. They have supported several students and researchers to achieve the goals of the META project. Our META group got extremely valuable feedback from our collaborators and beta testers about the ideas implemented in the OpenMETA tools.

The members of the META group have contributed immensely to my personal and professional time at Vanderbilt University. This group has been a source of friendships as well as good advice and collaboration. The work presented in this thesis is a work of the META group and several other teams. In particular, I would like to thank: Adam Nagel for his help with the design of the dynamics portion of the integration language Cyber-Physical Modeling Language (CyPhyML) and component specification; Patrik Meijer for his help in redesigning and reimplementing certain pieces of the OpenMETA tools and make it a more robust tool in general; Kevin Smyth for all the pair programming, debugging difficult problems, and sharing his programming experience; and the entire META development team.

Lastly, I would like to thank my family for all their love and encouragement. For my parents who raised me with a love of science and technology. I am grateful for all the support they gave me over the past decades. They always helped me to make my decisions which I greatly appreciate. Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AADL	Architecture Analysis and Design Language 8
ARDP	Analysis-driven Rapid Design Process 3, 55–58, 60, 71–73, 77, 80, 82–85
AVM	Adaptive Vehicle Make iii, 1, 2, 32, 33, 39, 43, 82
BPMN	Business Process Modeling Notation 22
CAD	Computer-Aided Design 26, 32, 33, 39, 50
CBD	Contract-Based Design 10–12, 17
CFD	Computational Fluid Dynamics 33, 50
CPS	Cyber-Physical System 1–3, 5, 12, 17, 21, 28, 38, 41, 48, 51, 55, 57, 80–83
CyPhyML	Cyber-Physical Modeling Language iv, 30, 37–39, 43, 44, 46, 82
DAE	Differential Algebraic Equation 20
DARPA	Defense Advanced Research Projects Agency iii, 1, 82
DASSL	differential/algebraic system solver 59
DBD	Dysfunctional Behavior Database 27
DC	direct current 20
DESERT	Design Space Exploration and Refinement Tool 40, 64, 65
DoE	Design of Experiment 32, 51, 60, 67, 70, 71, 78, 81
DSL	Domain-Specific Language 14
DSML	Domain-Specific Modeling Language 43
eCar	electric car 25, 26
ECU	Engine Control Unit 6
EPA	US Environmental Protection Agency 22
EPS	Electric Power System 58
FEA	Finite Element Analysis 26–28, 33, 39, 49, 50, 57
FMEA	Failure Modes and Effects Analysis 27

GME Generic Modeling Environment 43

GPL General-Purpose Language 14

HBGL Hybrid Bond Graph Language 44

KCL Kirchhoff's Current Law 20

LOC Lines-Of-Code 5

MBD Model-Based Development 8, 10, 25

MBSE Model-Based Systems Engineering 3, 5, 12, 13, 16, 21, 24, 25, 28, 30, 36, 37, 53, 56

MCS Monte Carlo Simulation 52

MDAO Multidisciplinary Design Analysis and Optimization 3, 27, 28, 33–35

MoC Model of Computation 4

ModelicaML Modelica Modeling Language 22, 23

MPP Most Probable Point Method 52

MSL Modelica Standard Library 20, 58, 71

OCL Object Constraint Language 40

ODE Ordinary Differential Equation 18, 47

OMG Object Management Group 14

OS Operating System 6

PBD Platform-Based Design 9–11, 16, 20

PCC Probabilistic Certificate of Correctness 32, 51, 52, 60, 67–69, 71, 77, 78, 81

PCE Polynomial Chaos Expansion 52

QoS Quality of Service 14

RML Requirements Modeling Language 24

SCAP Sequential Causality Assignment Propagation 18

SCR Software Cost Reduction 23

SD System Dynamics 57

SDD	Simulation-Driven Design 3, 25, 27
SoC	system-on-chip 10, 14
SPD	Sequenced Parametric Diagram 26
SysML	Systems Modeling Language 8, 14–16, 22–28
TGG	Triple Graph Grammar 26
TOPSIS	Technique for Order of Preference by Similarity to Ideal Solution 35, 65
TS	Taylor Series Approximation 52
UBML	Uniform Behavior Modeling Language 26, 28
UDR	Univariate Dimension Reduction Method 52
UI	User Interface 22
UML	Unified Modeling Language 8, 14, 15, 22, 37, 39, 49, 50, 53
XMI	XML Metadata Interchange 15
XML	Extensible Markup Language 15

CHAPTER I

INTRODUCTION

Product design has become increasingly complex in recent decades. Several design methods and processes have been developed to reduce the design complexity to a manageable level. These methods and approaches include: layered design, component-based design, the v-model, model-based development, virtual integration, platform-based design, and contract-based design. Model-based systems engineering leverages these methods to manage design complexity and to reduce development time and costs. In contrast to the traditional document-centric design process, model-based design approaches maintain traceability and dependency among artifacts in the form of relationships between models. The building blocks of a model-based engineering tool are models that represent the requirements, system components, and subsystems. Using model-based systems engineering tools has become an accepted practice in the industry.

Challenges

Most of these design methods and processes were primarily developed for either software or hardware development. However, in Cyber-Physical Systems, computational elements are tightly integrated with physical processes and physical components. Computational elements often interact with the physical system through a distributed network [87, 88, 131]. Physical systems are acausal systems, which means that they do not have predefined inputs and outputs by nature. For a specific operating mode of the system, inputs and outputs can be derived, but it is a cumbersome process for a large complex system. One challenge is how to effectively co-design physical and software components.

Customer expectations for product cycles are getting shorter [91]. Large software companies create software releases continuously [116]; sometimes more than one new release is published within a day. In software development, software releases and deployment are highly automated and often follow an iterative design process with continuous integration systems that run automated tests. By decreasing the design cycle time and thereby increasing the number of design iterations within a given period of time, designers can be more confident that any potential problems will be discovered. The design process for complex Cyber-Physical Systems (CPSs) has *not* reached such a fast pace *yet*. In 2010, the Defense Advanced Research Projects Agency (DARPA) initiated the Adaptive Vehicle Make (AVM) program to develop a fully integrated design process for complex CPSs [44]. The overall goal of the AVM program was to speed up the development time by 5x for complex CPSs. Designing complex CPSs involves a large number of discipline-specific models and analysis tools. Miscommunication between models and analysis tools has negative impacts on the development time and cost. Therefore, design processes for CPSs must be orchestrated in a modeling environment where this communication is captured from the early stages of the design process to the manufacturing of the system.

There are hundreds of distinct tools used in the automotive and aerospace industry to analyze different aspects of a complex system design. These tools include both in-house, i.e., internally developed and maintained, (70%) and commercial off-the-self tools (30%), which shows that there is no single tool that can deal with all aspects of a complex design problem [146]. An adequate model of Cyber-Physical Systems must: (1) capture domain interactions (e.g., electrical power and mechanical systems), (2) incorporate multiple aspects and domain models for each component, (3) support a wide variety of analysis techniques, (4) enable the reuse of existing models from libraries, and (5) extract sufficient information from model libraries to support architecture exploration for product families.

Problem Description

Consider the modeling of CPSs with heterogeneous component models and their interactions: each component model may consist of several domain-specific models that are related to each other. There are many relationships between those domain-specific models including physical or parametric relationships. Our goal is to develop concepts to capture integrated domain-specific models and their relationships for CPSs in a *Model Integration Platform*. When the domain-specific models are composed, a *Tool Integration Platform* will address how to operate domain-specific tools to facilitate domain-specific analyses driven by the composed domain-specific models. Usually, there is a large number of domain-specific analyses involved in a complex CPSs design problem. The required time to perform each execution can vary anywhere between a few seconds to several hours depending on the tool and the models. The challenge is further compounded when these analyses must be performed for multiple design variations or in an optimization loop. An *Execution Integration Platform* will address how to arrange large-scale analysis of models by the parallel execution of all independent analyses. Because the analysis can take several hours, even using parallel execution, a *Visualization Integration Platform* will address how to collect and visualize partial datasets as individual analyses are completed, before the full result set is available. We will develop a design process that presents how to use these four platforms to reduce overall design time.

Thesis Goals

We will contribute to three platforms to improve efficiency and quality of a design process focusing on high-level design of CPSs: a Model Integration Platform, a Tool Integration Platform, and an Execution Integration Platform. The Model Integration Platform will (a) use heterogeneous component models, (b) keep the multi-domain models consistent, (c) track model dependencies, and (d) facilitate importing models from existing libraries. The Tool Integration Platform will accommodate a variety of analysis tools with flexibility to add new tools in the future. The Execution Integration Platform will provide an analysis tool *independent* framework for analysis execution and organization of analysis results. In addition to these platform contributions, we will prototype an analysis-driven rapid iterative design process as part of this research.

Thesis Outline

This thesis is organized as follows: Chapter II gives an overview on Cyber-Physical Systems (CPSs), existing design processes, Model-Based Systems Engineering (MBSE), model interfaces and composition, requirement specification, Simulation-Driven Design (SDD), and Multidisciplinary Design Analysis and Optimization (MDAO); finally, it concludes with the lessons learned. In Chapter III the high-level design flow is presented along with four integration platforms: (a) the Model Integration Platform, (b) the Tool Integration Platform, (c) the Execution Integration Platform, and (d) the Visualization Integration Platform. Chapter IV discusses heterogeneous component models in detail. Chapter V presents the analysis templates and model execution framework. Chapter VI shows how the above-described concepts enable an Analysis-driven Rapid Design Process (ARDP) that was utilized on two use cases: (a) an oscillator design and (b) a ground vehicle driveline design. Finally, Chapter VII concludes this thesis.

CHAPTER II

BACKGROUND

In this chapter we present a summary of all relevant background information related to the scope of our research. A few key concepts are defined below that are used in this chapter:

Component is a design entity that represents a physical object, a hardware piece, or a software piece. Each component has interfaces that are used to communicate with other components and the environment.

Abstraction is an information reduction and conceptualization process with a set of assumptions for a software component, a hardware component, or a physical component.

Model views and model aspects consider a model with multiple interdependent representations of the same software component, hardware component, or physical component.

Heterogeneous models are models using (i) different modeling aspects, e.g., computational, behavioral, and structural; or (ii) different modeling domains, e.g., electrical, mechanical, etc.; or (iii) different Model of Computation (MoC), e.g., continuous time, discrete event, etc.

Multi-domain models are models that capture multiple physical domains and their interactions for the same component, e.g., an electrical component generates heat or an electrical component is temperature dependent.

Design is a collection of many interconnected components.

Design space is a collection of design alternatives or family of designs, where the alternatives differ from each other either in architecture or in design parameters.

Discrete design space considers only design alternatives from a design space where there is an architecture difference, or for a given architecture, alternative component options are available for substitution.

Parametric design space often assumes a fixed architecture choice and considers continuous parameter variations of components, subsystems, or both.

Virtual integration is a process in which models are composed and evaluated to predict the expected output of the system.

Cyber-Physical Systems (CPSs)

Cyber-Physical Systems (CPSs) are complex multi-domain systems that include highly interconnected and interdependent computational and physical components. CPSs orchestrate networked computational resources with physical systems [87, 88, 89, 131, 147]. Examples for CPSs are: avionics, transportation air traffic control, automotive, building systems, telecommunication systems, military systems (e.g., satellites), power generation and distribution, and factory automation. In CPSs there are several challenges to be solved which involve scheduling problems, capturing software and hardware interactions, representing physical interactions, and solving control problems.

For instance, in the power generation and distribution domain, one challenge is how to couple and operate green energy and continuous energy sources in the same network. Green energy is usually available based on a state of the environment, e.g., direct sunlight to solar panels, or wind rotating wind turbines. Another challenge is that the power demand is increasing and decreasing based on the time of the day, but the produced green energy is often not stored, therefore, it must be consumed.

Another example is the automotive domain: a premium car today contains at least 80 computers including: Engine Control Unit, Transmission Control Unit, Automatic Braking System, Air Conditioning system, entertainment system, navigation system, etc. All computers are connected to the car's system bus; currently, the controller area network (CAN) bus is the standard (in the near future the standard will be FlexRay). There are around 100 million Lines-Of-Code (LOC) for such a car.

These systems often contain multiple domains and their interactions such as electro-mechanical or electro-hydraulics. Physical interactions are replaced by software interactions, which are more flexible, but it is harder to understand and model them. Physical systems are increasingly segregated by software components, e.g., fly-by-wire systems in avionics. This trend increases the demand on Model-Based Systems Engineering to succeed with providing comprehensive and complete tool sets and design processes that are capable of representing and analyzing complex CPSs.

Existing Design Processes

Layered Design

A design process using layered design defines abstraction levels for software products [91]. Every abstraction level is associated with different design activities. These abstraction levels help to decompose and compartmentalize a complex design problem into smaller units. The layered design process can have many abstraction levels, which are called layers. A lower-level layer does *not belong to* a higher-level layer, rather it is *used by* the higher-lever layer. As a result, a layered design has a compound relationship of the *uses* form instead of the *part of* form between layers. By defining multiple layers for an application domain problem, the various design concerns are separated. Each layer is built on top of another layer. A lower-level layer encapsulates design concerns and implementations and hides them from the higher-level layers. This layered design approach was adapted for various application domains including automobile and avionic domains.

For example, in the automobile domain the AUTOSAR [9] standard was developed to define several abstraction layers: the Microcontroller Abstraction Layer, the Engine Control Unit (ECU) and Complex Drivers Layer, the Services Layer, and the Application Layer. The software components implemented in the Application Layer communicate directly with the Operating System (OS) and Services Layer; whereas the lowest level, the Microcontroller Abstraction Layer, encapsulates the actual microcontroller implementation. In the avionics domain, similar abstraction layers are defined by the ARINC standards [5].

One of the challenges in layered design is to define a set of abstraction layers for a domain problem which provide sufficient information and encapsulation for the higher-level layers, while the number of implementation options remains broad.

Component-Based Design

In component-based design, designs are composed of multiple design entities called *components* [38]. Each component has a set of interfaces that are used in the composition, where each component interface may be connected to one or many other component interfaces. Components encapsulate the implementation, which makes them highly reusable throughout the design process. Having components defined can significantly accelerate the design process, because the components can be developed and implemented in parallel. Defining components on different levels of the design containment is called hierarchical system decomposition; this reduces the complexity of the design problem [121].

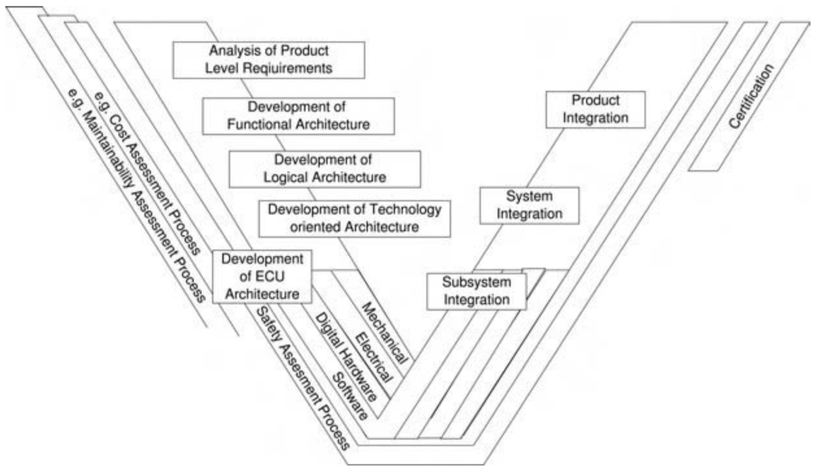

Analysis of Product Level Reqiurements
Development of Functional Architecture
Development of Logical Architecture
Development of Technology oriented Architecture
Development of ECU Architecture
Mechanical
Electrical
Digital Hardware
Software
Safety Assesment Process
e.g. Cost Assessment Process
e.g. Maintainability Assessment Process
Subsystem Integration
System Integration
Product Integration
Certification

Figure 1: V-model [133]

There are two major challenges with component-based design: (a) how to specify component interfaces to be rich enough to cover all phases of the entire design process and (b) how to capture a family of designs with various alternative component and architecture options possibly representing product variations and product families.

V-model

The V-model is a product development process primarily developed for software applications. Originally, the V-model was published in 1997 in the Development Standards for IT Systems of the Federal Republic of Germany document [41]. The V-Model became a standard for all civil and military federal agencies in Germany. The objectives of the V-Model are: minimization of project risks, improvement and guarantee of quality, reduction of total cost, and improvement of communication between stakeholders. Since the initial development of the standard, the V-model was adapted for other application domains including aircraft design and software design. The V-model is named after its V-shape diagram, as shown in Figure 1, which splits the development process into two major phases: (a) a design phase and (b) an integration phase. [49]

The design phase starts with the analysis of the product-level requirements, which is followed by the development of a functional architecture diagram. The functional architecture diagram is divided into individual domains such as mechanical, electrical, digital hardware, software, etc. After the individual domain-specific components and subsystems are designed and tested, they are integrated at the subsystem level followed by the system-level integration across multiple domains according to the logical architecture diagram. Product integration, validation, and product certification against product level requirements are the final steps in the integration phase of the V-model.

The V-model reaches its limitations as systems become more and more complex, in particular when they involve several domains along with cross-domain interactions and interdependencies. Because the domains are kept in separate models and there are no direct linkages between the different domain representations, the subsystem- and system-level integrations are tedious. After the system is physically integrated, it may fail to meet product level requirements. Such failures are identified *too late* in the design process, and they necessitate model updates or in a worst case scenario in a complete redesign of the system. All the aforementioned shortcomings of the V-model inject costly unpredictable delays in the entire product development process.

Model-Based Development

In Model-Based Development (MBD), models are used to capture and organize information for products to be designed [122, 139]. A collection of models is a full representation of a system design. Models and their relationships are explicit representations of traceability and dependency of design artifacts. The MBD design process captures requirements from the early stages of the design activity in the form of models. It is a common practice in MBD to use code generation tools to facilitate virtual integration and to automate portions of the subsystem- and system-level integration as well as verify the design against the specified design requirements. The MBD design process inspired system engineers to create different modeling languages such as the Architecture Analysis and Design Language (AADL) [46] for performance critical system design, the Unified Modeling Language (UML) is a general purpose modeling language for the software engineering field, and the Systems Modeling Language (SysML) for system-level modeling.

Using system-level modeling languages does not solve the entire product design problem. Therefore, application and domain-specific tools must be used in conjunction with system- and architecture-level modeling languages and tools. Tool-specific models are developed for different aspects (views). For instance, MATLAB-Simulink [96] is often used for controller and control-flow modeling; 20-Sim [1], Bond Graphs [74], MATLAB-SimScape [94], Catia [143], and Modelica [7] are used for physical system modeling. Automatic code generation for simulation and behavior analysis is possible, which helps to verify the design against the requirements and validate the models against empirical data.

The drawback to having tool-specific models is that it increases the risk of inconsistency between different aspects (views) of the same design entity.

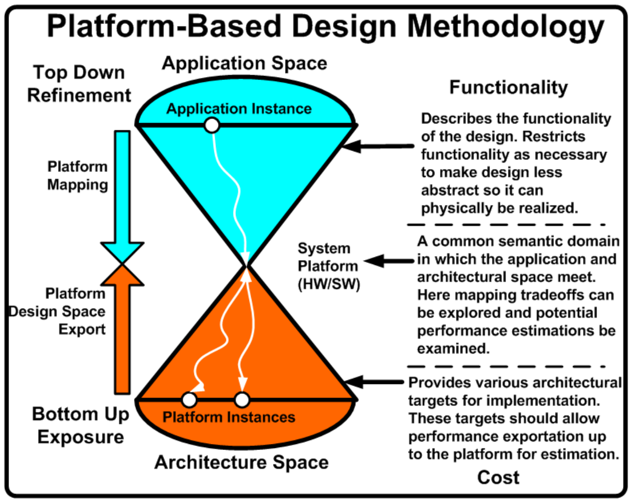

Platform-Based Design Methodology
Top Down Refinement
Application Space
Functionality
Application Instance
Describes the functionality of the design. Restricts functionality as necessary to make design less abstract so it can physically be realized.
Platform Mapping
System Platform (HW/SW)
A common semantic domain in which the application and architectural space meet. Here mapping tradeoffs can be explored and potential performance estimations be examined.
Platform Design Space Export
Provides various architectural targets for implementation. These targets should allow performance exportation up to the platform for estimation.
Bottom Up Exposure
Platform Instances
Architecture Space
Cost

Figure 2: Platform-Based Design's general framework [75]

Virtual integration

Historically, systems integration (e.g., V-model integration phase) is a challenging task because after the system prototype is physically integrated and realized, it still may fail to meet system requirements. Virtual integration addresses this challenge by composing models of components and subsystems, resulting in a fully-composed system model. We can then perform analyses on this virtual prototype to verify the predicted behavior of the system. Virtual integration minimizes the risk in the integration phase of the V-model, because the designed subsystems can be virtually composed and analyzed at the system level according to the system architecture diagram. As a result, this process reduces the number of physical prototypes required to test the system.

As we discussed before, a complex system design often involves several domains including multiple physical domains such as electrical, hydraulic, one- and multi-dimensional mechanical, and thermal. Therefore, the modeling tools should support heterogeneous component models and modeling their interactions. For instance, Ptolemy [24] addressed heterogeneous component modeling and composition with different semantics. Modelica [7] and SimScape supports physical modeling and simulation at an advanced level including multi-body dynamics and hardware-in-the-loop simulations.

Platform-Based Design

Platform-Based Design (PBD) is an integration-oriented design approach for developing complex products based upon a software or a hardware platform [10, 75]. The software or hardware platform provides a platform-specific abstraction layer for the designers to develop platform-specific applications. A common semantic domain, often called the system platform, is presented in the middle of Figure 2 [75]. The application space is mapped to the system platform by a top-down refinement process and the architecture space is exported to the system platform by a bottom-up exposure.

PBD emphasizes the reuse of platform-specific components and is intended to reduce development costs, risks, and time by using virtual integration [10]. It incorporates the advantages of four design methods: (1) Component-Based Design, (2) virtual integration, (3) layered design, and (4) Model-Based Development. By combining these four design methods, PBD reduces the design complexity by hierarchical system decomposition and definitions of platform-specific abstraction layers. In addition to the decomposition techniques, PBD also supports multi-layer optimization through multiple viewpoints, whereas other design processes do not. The PBD concepts have been successfully applied in several different domains including: automotive, building automation, printers, wireless networks, network processors, system-on-chip (SoC) and electronic design. One key difference between the layered design process and PBD is that layered design predefines the abstraction layers for *any* design problem within the domain, but PBD leaves the abstraction layer definitions to the particular application or design platform.

Each design platform consists of three key elements: (a) a component library, (b) models of components, and (c) connectivity and composition rules between components. The models of components may be mathematical models at different levels of abstraction or placeholders to indicate customization. By adding platform-specific constraints to the composition of components, a family of designs is represented which satisfy those constraints. A platform instance is called an architecture, which is defined by the composition of platform components. An architecture captures how the system does what it is supposed to do and fulfills all platform-specific constraints.

Contract-Based Design

Contract-Based Design (CBD) augments Platform-Based Design with *contracts*, where a contract is defined for each component with a set of *assumptions* and *promises* [22, 133]. The assumptions are defined *only* for ports that are controlled by the environment. The promises are defined *only* for ports that are controlled by the component. There are two types of contracts: *horizontal* and *vertical*. *Horizontal contracts* are defined among subsystems on the same level of abstraction, and are used for virtual integration testing. *Vertical contracts* are used across design layers, when each component can be further refined or decomposed in the design process.

Assumptions are frequently categorized into strong and weak assumptions. Examples for strong assumptions are: meeting a standard or being part of vertical contracts for composition. Violation of vertical contracts means incompatibility across design layers in the decomposition hierarchy. Weak assumptions are desired properties of the system such as cost, manufacturing lead time, etc. When a design becomes invalid and fails to meet the contracts, it is possible to backtrack the contract assertion to the higher-level contracts by maintaining dependency between contracts. In such cases, weak assumptions can be relaxed (e.g. rebudgeting) to fulfill all contracts.

CBD augments PBD with contracts; these contracts assist with determining whether the architecture composition is valid or not. CBD considers only component choices from the component library that result in *legal* composition during architecture exploration.

Summary

In this section we presented existing design processes and methods. The goals of each design method are to reduce design complexity and development time. Design complexity is reduced by defining abstraction layers or decomposing the design problem based on containment. Development time is often reduced by using components or models as design entities. Components and models represent the designed product with an explicit representation of traceability and dependency of the design artifacts. These components and models are highly reusable in the design, even across multiple subsystems. Components and models can be virtually integrated using composition which reduces the number of physical prototypes that must be built and tested.

When the design product is based on a specific platform, a platform-specific abstraction is provided for the designers to build their platform-specific components and products. The platform can be either a software platform, a hardware platform, or both. The platform-specific components and the platform itself can have a set of assumptions and promises. The set of assumptions and promises for a given component is a contract. Contract-Based Design defines the types and the composition of contracts. Contracts can be used to verify the correctness of the virtual integration. If there is an assertion on a system-level contract, it is possible to trace back to a lower-level component in the hierarchy that caused the assertion.

Evaluation

The design processes outlined above were mainly developed for software design, where models and components have inputs and outputs. This means that the direction of information flow between components (i.e., causality) is explicitly defined in the system by the designers.

In complex Cyber-Physical System design, components often span multiple domains including behavioral, structural, and computational. This requires a design process that accommodates multiple views of component models. The different views are often dependent on each other. The design process must support acausal (i.e., bidirectional) interfaces. Depending on the operation mode of the system, the inputs and outputs may change. Consider a hybrid car: when the car accelerates the batteries provide electrical energy to the mechanical system; when the car applies the brakes the mechanical system charges the batteries.

In Contract-Based Design the promises are defined only for ports that are controlled by the environment; this design approach is not directly applicable for physical system modeling where the inputs and the outputs can change. For CBD to be applicable, each operation mode of the system must first be elaborated. Unfortunately, elaborating all operation modes of complex CPSs results in a large combinatorial space, which is difficult to manage and simultaneously increases model complexity. One of the goals of many design processes is to *reduce* complexity.

The aforementioned design methods give no or minimal support for architecture and parametric design space exploration which highly limits designers in finding the optimal solution for their problem. It is important that any evaluation method used in the design process (e.g., evaluation of contract assertions) can also be applied in parametric design space exploration (e.g., in a design optimization).

Model-Based Systems Engineering

Model-Based Systems Engineering (MBSE) is a systems engineering methodology that captures views and analyses for a product design at the system level. The system-level views are the collection of virtually integrated models that represent the integrated system [47, 139, 141].

In traditional engineering design, several documents are developed: requirement documents, conceptual design documents, detailed design documents, etc. Unfortunately, there is an inherent dependency across all aforementioned documents. In this document-centric approach, the documents are the primary artifacts in the design process. The relationships between such documents are often poorly maintained. However, MBSE organizes the documents and their dependencies by associating them to models. These models become the primary artifacts in the design process and are stored in a model database. The models refer to each other explicitly and the relationship between models yields the relationships between documents. When a document is changed it results in a model change. By this model change all relevant linked models and other documents can be identified and updated.

The goal of Model-Based Systems Engineering is to improve efficiency and quality of the design process and semantic interoperability. However, several obstacles remain in achieving that goal [120].

Obstacles to improving efficiency

Manually developing models is labor intensive and expensive. MBSE tools should leverage and accommodate pre-existing models and model libraries. Complex design problems often involve the usage of several heterogeneous model libraries and analysis tools. Setup of analyses is time-consuming and cumbersome due to the lack of the interoperability of the models. Having heterogeneous analysis tools implies having heterogeneous models, which leads to multiple views (aspects, concerns) of the same model. Manually maintaining the dependencies between model views of the same system is error-prone. Even if the models are stored in a centralized place, there are recurring activities, e.g., writing design reports, which can negatively impact the efficiency unless some or full automation is provided by the MBSE tools.

Obstacles to improving quality

The different models and model views must be kept synchronized with each other to ensure model consistency. Assume that a model has at least two views: a behavioral and a structural view. Some of the model parameters are shared between the different views. Changing such interdependent parameter values *must* result in a change in *both* analysis-specific behavioral and structural models.

Many MBSE tools provide hierarchical modeling, which aids to group information and knowledge that people can easily process in every level of the hierarchy. In complex system designs, multiple teams are involved in the entire design process, where each team may include several people. Hierarchical modeling improves and clarifies communication between people and teams. Different people and design teams have different objectives, beliefs, and preferences which form the basis of their design decisions. Those divergent design decisions may lead to irrational designs.

Domain-Specific Languages (DSLs)

A General-Purpose Language (GPL) is usually a general purpose programming language which is applicable in a wide variety of application domains. This is in contrast to Domain-Specific Languages (DSLs), which are specifically built for a given domain and provide domain-specific abstraction through the use of terms and concepts from a specific domain [150, 51]. Oftentimes, several custom tools are developed in conjunction with the DSL. Generally, these tools are called *interpreters* that act on the domain-specific models, and *generators* (i.e., compilers) that translate the domain-specific models into source code or another model representation.

Each DSL consists of one or more *concrete syntax* definitions and one *abstract syntax* definition. The concrete syntax is used to visualize, manage, and edit domain-specific models; it can be graphical, textual, tabular, or a mix of those. The abstract syntax of a DSL is defined in a meta-model, which enforces the static or structural semantics of the domain. The meta-model contains type definitions, composition rules, and constraints. However, the execution or behavioral semantics of a DSL are defined and realized by an execution engine. Different partitions of the DSL can have different behavioral semantics and could require multiple execution engines to realize them. DSLs are developed for a variety of engineers, and not just software engineers, depending on the application domain [150].

Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a graphical modeling language developed by the Object Management Group (OMG) [50, 98, 115]. UML is used to model software application structure, behavior, and architecture. Different application domains may require more specific concepts than what UML specifies. UML profiles are extensions to UML that define stereotypes using domain-specific concepts; these stereotypes can be used to build domain-specific models. UML profiles have been developed and maintained for commonly used application domains. For example, a few domain- or application-specific profiles are: UML Profile for Software Radio, UML Profile for Modeling Quality of Service (QoS) and Fault Tolerance Characteristics and Mechanisms, UML Profile for system-on-chip (SoC), and UML Testing Profile. The first version of the UML Profile for Systems Engineering (SysML) was developed by systems engineers in 2005, who aimed to model systems engineering applications including physical and computational parts.

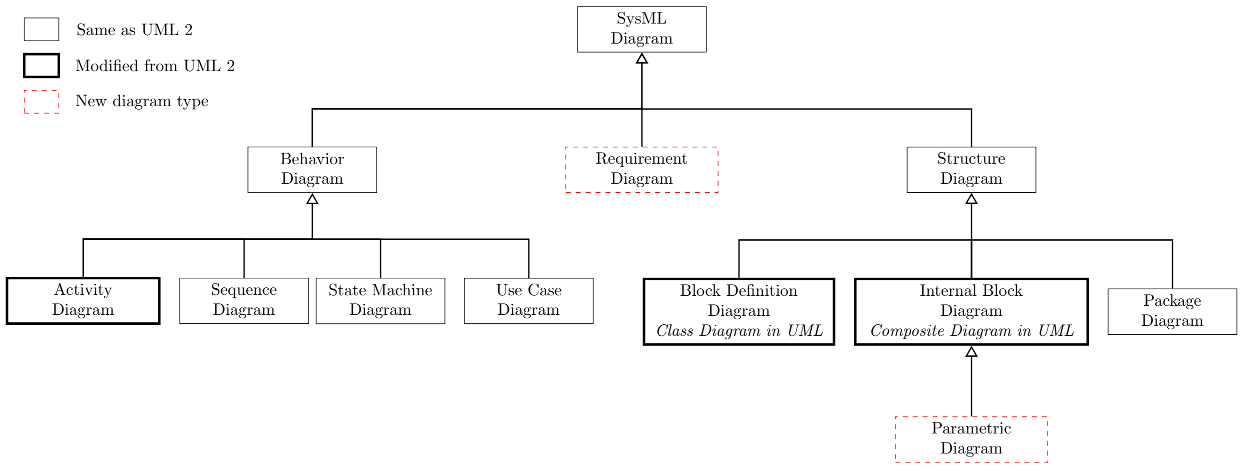


Figure 3: SysML taxonomy diagram

Systems Modeling Language (SysML)

The Systems Modeling Language (SysML) is a general purpose graphical modeling language for systems engineering applications [52, 141, 152]. SysML extends and modifies a subset of the UML 2.0 as shown in Figure 3. The UML *Activity Diagram* contains some extensions and restrictions in the SysML profile. The *Block Definition Diagram* is similar to the UML *Class Diagram* and the *Internal Block Diagram* is similar to the UML *Composite Structure Diagram*; both have some extensions and restrictions as described in the SysML specification [114].

The new diagram types (i.e., extensions) are: the *Requirement Diagram* and the *Parametric Diagram*. Each SysML and UML model can be serialized into XML Metadata Interchange (XMI) format for model exchange between different SysML or UML tools, respectively.

The block definition diagram and the internal block diagram provide support for nested and typed ports. The nested and typed ports enable reusable blocks and clearly defined interfaces for composition. SysML defines two kinds of ports: one that exposes its own features called full ports and another one that exposes ports or features of its parent called proxy ports. Another modification is the support of flow properties in block diagrams. The flow properties can define the kind of items that flow through the ports, for instance: data, material, or energy. Each block can define a set of flow properties that are associated with ports. Definition of flow properties for ports is essential for physical system modeling and developing interconnectable components. In older versions of SysML (before version 1.3), flow ports served the same purpose; the newer versions of the SysML specification describe how to migrate the old models, because the flow ports are marked as deprecated elements.

SysML supports text-based requirements modeling and organization using requirement diagrams. The requirement diagram supports hierarchical decomposition or grouping of the requirements. Each requirement may specify a function or a performance goal that the system must perform or achieve, respectively. The requirement block can be related to other blocks and elements in the SysML model using the *copy*, *derived*, *satisfy*, *verify*, *refine*, and *trace* relationships.

The parametric diagram is a specialization of the internal block diagram. Each parametric diagram may contain one or more constraint properties and their parameters. In addition to the internal block diagram restrictions, all properties must be bound to either a constraint parameter or contain a property that is bound to one. Internal block diagrams are similar to the placeholders in the PBD. A parametric block diagram with constraints and bound property values is similar to a platform instance in the Platform-Based Design.

Several extensions have been developed for SysML including a collaborative web-based SysML editor with limited authoring capabilities [21]. Other SysML-based tools and extensions are described in detail in the Simulation-Driven Design section.

Evaluation

Domain-Specific Languages provide strong type checking for model composition using domain-specific concepts and terms. SysML provides a general-purpose Model-Based Systems Engineering modeling language with built-in model traceability and dependency, but it lacks structural semantics and domain-specific concepts. Domain-specific concepts can be added to SysML by defining SysML profiles. Many application domains have been developed as SysML profiles because the SysML specification [114] does *not* define any semantics for general SysML models. However, when a SysML profile is used and connections or objects are created the structural semantics are *not* strictly enforced at the time of modeling. Consider a SysML profile defined for a domain which contains mechanical and electrical port types. A designer is allowed to connect a mechanical port to an electrical port, but that connection makes the entire model invalid, because the model cannot be physically realized. Such modeling errors might be revealed too late in the design process which could increase the design time and costs.

Model-Based Systems Engineering tools must support: (a) importing existing multi-domain component libraries for each aspect of the component including physical behavior, structural, and computational; (b) strong type checking for composition (i.e., structural semantics) which makes the models correct by construction; (c) seamless integration with existing analysis tools for automated analysis execution; and (d) iterative model development.

Model Interfaces and Composition

Models interact with other models or the environment through model interfaces. Models are composed into larger entities by connecting one or more model interfaces together. The interfaces are used to exchange data, material, or energy between components or to establish a structural relationship between components [42]. Strongly-typed interfaces improve model clarity and eliminate composition mistakes, e.g., connecting a data interface to an energy flow interface. The interfaces can be divided into two main categories: causal and acausal. Causal interfaces define the direction of information flow through the interfaces; hence, they are either inputs or outputs. When causal interfaces are composed, there are strict rules describing how inputs and outputs can be connected. Acausal interfaces do not predefine the direction of the energy flow; the direction is usually determined by the composition or the operating mode of the system. If the operating mode changes during the analysis of the system, the inputs and outputs might need to be changed for certain interfaces.

In Contract-Based Design, contracts are associated with component models, where each contract is given as a set of promises and assumptions defined on input and output interfaces, respectively. If the environment fulfills the specified assumptions, then the model promises certain properties or behavior. The assumption/promise approach is also called the assume/guarantee, or assumption/commitment approach. This paradigm has been applied in a number of approaches including software specification, system specification, and logical proofs of system-level properties. These component-based contract specifications have been generalized to architectural contracts for discrete event systems and state machines [22]. It is important to note that both discrete event systems and state machines have causal interfaces.

Contract specifications can be used for CPSs as presented in [133, 112]. In CPSs, contracts are always specified in the computational blocks or they are defined on the boundary of the physical system. The physical system model interacts with computational blocks through causal interfaces using sensors and actuators. Contracts are never specified on acausal interfaces inside a physical system model or between physical component models.

Physical system models require acausal interfaces for efficient modeling and design. Several modeling languages support acausal or equation-based modeling such as Bond Graphs [74], Modelica [7, 53, 149], and SimScape Language [95].

Bond Graphs

The Bond Graph notation is a graphical modeling language for physical systems based on power and energy flow. The Bond Graph elements are domain *independent*, which aims to bridge the communication gap between different domain experts. In other words, if a mechanical engineer and an electrical engineer are both familiar with Bond Graphs, they can model electro-mechanical systems including mechanical systems or electrical circuits, respectively. When they share the models with each other, both of them can interpret the behavior of entire system including cross-domain interactions. This property of Bond Graphs provides reusability and cross-domain interoperability.

Basic Bond Graphs contain three kinds of elements: one-port elements, two-port elements, and multi-port elements called junctions. The ports of the elements are connected together and the connections are called *bonds* and are represented with a half-arrow. The direction of the half-arrow denotes the assumed positive direction of the power flow between the adjacent elements. Each bond represents an *effort* and a *flow* variable, which are domain dependent as shown in Table 1. The product of the effort and flow variables is power, which corresponds to the energy flow between the adjacent elements.

Physical Domain	Effort	Symbol	Unit	Flow	Symbol	Unit
General	e			f		
Mechanical translational	force	F	N	linear velocity	v	m/s
Mechanical rotational	torque	τ	N m	angular velocity	ω	rad/s
Electrical	voltage	V or u	V	current	I or i	A
Hydraulic	pressure	P	Pa	volumetric flow rate	Q	m^3/s
Thermal	temperature	T	°C	entropy flow rate	S	W/°C

Table 1: Bond graph variables for physical domains

Even though the bonds are directional, Bond Graph is an acausal modeling paradigm, which means that there are no dedicated input and output variables in the system. Inputs and outputs depend on the graph and the composition of the elements. Causality is assigned to each bond based on a given set of rules for each element using the Sequential Causality Assignment Propagation (SCAP) algorithm [74]. Every valid Bond Graph can be directly translated into a set of Ordinary Differential Equations (ODEs) after the causality is assigned. The acausal modeling eliminates the need of rewriting the ODEs if the system topology changes, because the new set of ODEs can be *automatically* derived based on the new system's topology.

Bond Graphs are often used to model existing system dynamics to better understand, track, trace, and analyze fault propagation in physical systems [20, 45, 101, 102]. Bond Graphs are also frequently used for modeling and simulation of mechatronic systems [18]. The Bond Graph methodology [19] was augmented with multi-bond graphs to facilitate the modeling of complex multi-body dynamics [30]. In addition to the modeling and simulation of the mechatronic systems, the control of mechatronic systems is presented in [74].

Bond Graphs *do not* support hierarchical modeling, which can lead into diagrams with hundreds of elements and connections on the same diagram for a large scale complex system. Even though Bond Graph supports acausal models, it is challenging to compose Bond Graphs with other acausal (or equation-based) modeling languages such as Modelica. Designers like to reuse the existing model libraries even if they are built within a different modeling language. Basic Bond Graph modeling evolved to model complex multi-body dynamics [30], but complexity of the diagrams significantly increases, which reduces productivity and efficiency. To overcome the increasing model complexity, a Bond Graph library was developed for Modelica [29]. The Bond Graph library leverages the hierarchical decomposition feature of Modelica to manage design complexity at different levels of the hierarchy.

Bond Graphs have been applied to other domains. For example, the Modelica Bond Graph library is used to create the system dynamics model of a servo-positioning system; the power flow information is used to monitor the effectiveness of a control system [97]. The method compares the efficiency of controllers with different topologies, where one controller is a linear controller and the other one uses a non-linear anti-backlash element. A Modelica based simulation tool, Dymola, is used to generate the simulation results, which allows us to compare the efficiency of the two controllers in a quantitative way by using an energy-based control evaluation method. This approach helps control engineers to quantify the quality of a particular control design and compare different designs.

Modelica

Modelica is an equation-based unified object-oriented modeling language for systems modeling defined by the Modelica Language Specification [7]. The Modelica language is domain independent, but domain-specific libraries and concepts have been developed [53, 149]. In fact, a Modelica Standard Library is provided, which is built on the Modelica language, and provides domain-specific ports and elements for several physical domains such as electrical, mechanical rotational, mechanical translational, fluid, magnetic, thermal, etc. In Modelica everything is a *class*, and there are specialized classes which pose additional restrictions on the containment or the structure of the class. A few commonly used and important class types: *type*, *record*, *connector*, *model*, *block*, *function*, and *package*.

The *type* is used to specialize variables by restricting the causality (input or output), the size (e.g., 1-dimensional or 3-dimensional), and the unit (e.g., acceleration in m/s). The *record* is used to group multiple *type* definitions. For instance, different fluid types have different values for density, dynamic viscosity, kinematic viscosity, conductivity, etc. If all of those properties are grouped into a *record* it is easy to create several fluid type specifications like water, air, or carbon dioxide by providing the expected values.

The *connector* class type defines pairs of *potential* and *flow* variables as acausal ports. The flow variables are marked with a keyword called *flow*, whereas the potential variables are not marked explicitly. For instance, in the electrical domain the potential variable is the voltage and the flow variable is the current. This is similar to the Bond Graphs as discussed in the previous section. *Connectors* are connected with the *connect* statement. When the Modelica code is translated to a set of Differential Algebraic Equations (DAEs) the connect statements are resolved into multiple equations. For instance, in the electrical domain the set of connected *connectors* will be resolved to and obey Kirchhoff's Current Law (KCL).

The *model* and *block* are structurally similar except the *block* classes cannot contain any *connectors* (i.e., acausal interfaces). The *function* class cannot have any *connectors* and equations, it can only have inputs, outputs, parameters, and algorithms. All of the aforementioned elements can be organized into *packages*.

Modelica supports abstract classes by using the *partial* keyword and inheritance by using the *extend* keyword. Each class can be marked as *partial*, which means the class itself cannot be instantiated, because it does not contain the full definition yet. Each class can be extended (or derived) from another class that has at least the same class restrictions. For instance, a model can be extended from a block, but a block cannot be extended from a model. In addition to abstract classes and inheritance, templates and constraining classes on templates are also supported. The *replaceable* keyword identifies placeholders in the model similar to the placeholders in Platform-Based Design, where each placeholder can be constrained by an other class. When the placeholder is replaced with a class, that class must be the constraining class or any of its subtype.

Even though Modelica is a textual language and the Modelica code is stored as a text file with a .mo file extension, the Modelica Specification allows to store graphical information including locations and icons as annotations as well as tool-specific annotations. Modelica is a language and there are existing Modelica tools that can interpret the Modelica code and execute Modelica simulation models. The Modelica tools often contain a graphical editor and always contain at least one DAE solver. There are open-source Modelica simulation environments such as JModelica.org [100], and OpenModelica [117]. There are commercial Modelica simulation environments such as Dymola [144], MapleSim [93], Wolfram System-Modeler [154], SimulationX [70], and CyModelica [35].

Figure 4 depicts a Modelica model from the Modelica Standard Library (MSL) called `Modelica.Electrical.Machines.Examples.DCMachines.DCPM_Cooling`. The system contains a direct current (DC) motor with a cooling system, where the mechanical load of the

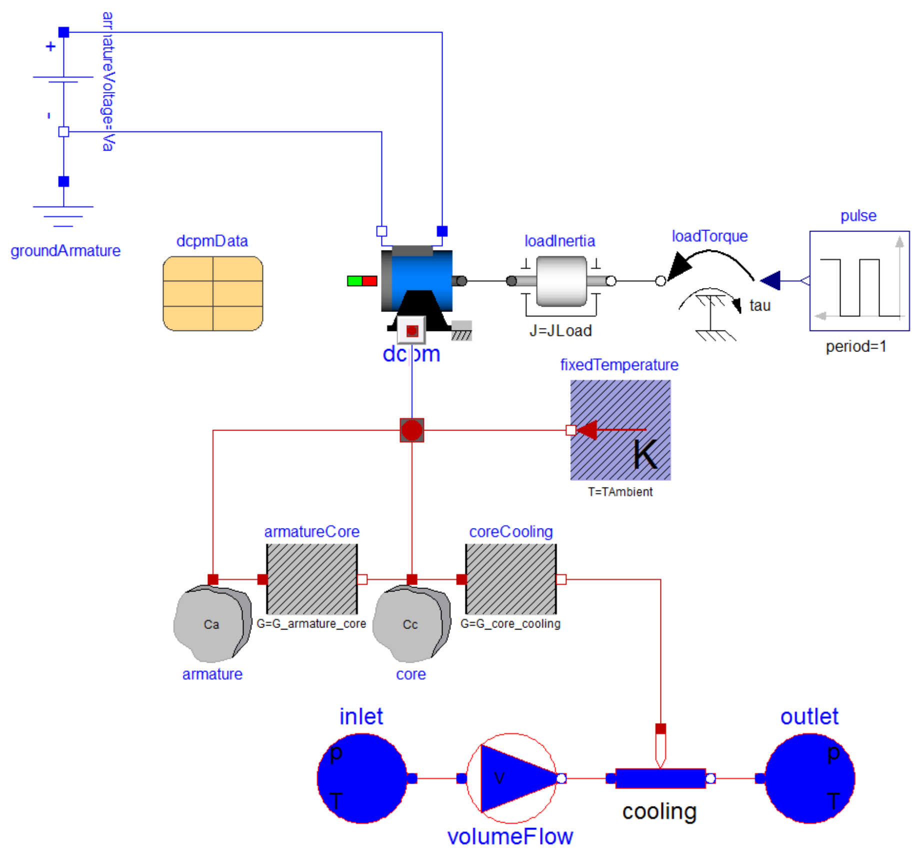


Figure 4: Example Modelica model

motor is pulse modulated. Even this simple system contains four different physical domains: electrical, mechanical rotational, thermal, and fluid. Each domain can be easily identified in the diagram, because the connectors are strongly typed and have different visual representations in each domain. These diagrams help to increase productivity and model clarity, and result in minimizing miscommunication between subject matter experts.

Evaluation

Strongly-typed causal and acausal interfaces are essential parts of Model-Based Systems Engineering for Cyber-Physical System design. They assist the designers with developing models that are correct by construction. For example, in Modelica it is *not* possible to connect an electrical pin to a mechanical flange (such a connection would result in compile time error and some editors prohibit such a connection in the graphical user interface). Another example is the fluid port connections: two fluid ports can be connected because they are structurally compatible, but the Modelica compiler also checks if the defined medium type matches on each connected port. This means that the system model will *not* be interpreted and simulated unless it is semantically correct.

Having well-defined domain-specific interfaces helps with building model libraries around partial (i.e., placeholder) models, which define the interface of a specific component while leaving the implementation undefined. These placeholders make architecture and design space exploration easier. The component placeholders can be easily substituted with any compatible component. With this flexibility, highly reusable and extensible component libraries, component interfaces, and architecture templates can be developed.

Requirements

Requirements are expected properties of the system and they are typically testable [126, 140]. There are two main categories of requirements: (a) functional requirements and (b) non-functional requirements. Functional requirements describe specific behavior or functionality of the system, i.e., what the system should do. Examples for functional requirements are: the system must minimize injuries upon collision or the system must transport people and objects from A to B. Non-functional requirements describe the desired operations of the system, i.e., how the system works or operates. Examples for non-functional requirements are: the safety system must be deployed when a collision is detected or the fuel consumption must be greater than 30 MPG over the highway US Environmental Protection Agency (EPA) drive cycle profile.

Several tools have been developed to model requirements such as Blueprint [17], Enterprise Architect [90], HP Quality Center [119], IBM Rational DOORS [64], IBM Rational DOORS Next Generation [65], Jira [8], PTC Integrity [129], Mingle [155], etc. The requirements management tools allow designers to store and retrieve design requirements written in a textual form as well as relate requirements to each other or group multiple requirements together. Some of these requirement management tools are coupled with requirement development, project management, (automated) testing, User Interface (UI) mockup, or visual modeling capabilities. Visual modeling is often provided as graphical models such as UML, Business Process Modeling Notation (BPMN) [113], or SysML models.

SysML provides a first class concept to support requirements in the form of requirement diagrams as presented in the previous section. Even though requirements are organized in models and several relationships can be established between the model entities, the requirements are still stored as textual artifacts without any execution semantics. To evaluate the requirements, execution semantics must be defined for the models or specific diagram types.

For instance, the Modelica Modeling Language (ModelicaML) [127] extends SysML with two new diagram types and modifies three existing SysML diagram types. These extensions

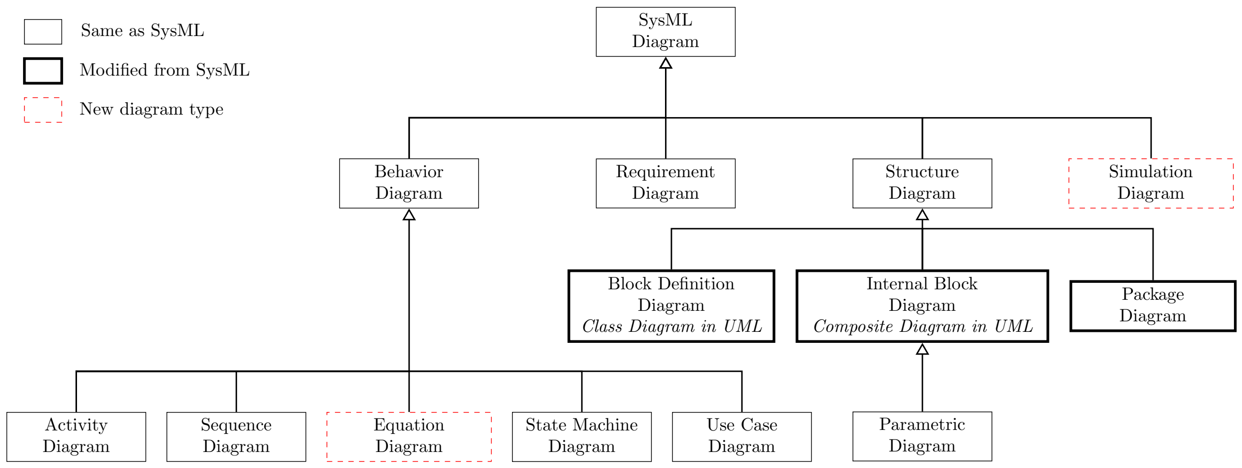


Figure 5: ModelicaML for SysML taxonomy diagram

and modifications facilitate the execution and evaluation of requirements for physical systems. Figure 5 depicts the ModelicaML taxonomy that includes the *equation diagram* and the *simulation diagram* in addition to the SysML taxonomy.

Modelica supports object-oriented equation-based modeling of system designs. However, the Modelica language itself lacks first class requirement support, thus ModelicaML was developed as a profile for SysML. ModelicaML augments SysML with an equation diagram that contains Modelica equations, which represent the model behavior, therefore it is a subclass of the behavior diagram. The simulation diagram defines an experiment for a referenced Modelica model including parameters for the included Modelica model, and storing simulation results called *simResults* for plotting and requirement evaluation. One or more requirements can be referenced from the simulation diagram, which are linked to the simulation results through the *satisfy* relationship.

Formal notation of requirements is used along with automated tools to produce high-quality requirement specifications. One formal notation of requirements is called Software Cost Reduction (SCR) [59], which was developed to concisely and unambiguously specify software requirements for real-time embedded systems. The SCR method uses tables to capture, relate, and organize design requirements that can be used to define design requirement specifications. If requirements are defined using formal notation (e.g., SCR) then requirement analysis can be performed including automated consistency checking of requirement specifications [58]. The automated consistency checker is used to check the specification in an application-independent way for reachability, coverage, syntax, and type correctness. The formal requirement specification is executed symbolically to verify that it represents designers or customers intent. An SCR* toolset for specifying requirements was developed [57] and applied in developing high assurance avionics systems [16]. Other research describes the benefits of formal Requirements Modeling Language (RML): develop and present requirements as models using object-centered representation and reasoning with models through consistency checking or simulation [55].

Requirement and design trade-offs

When a set of requirements is defined, there is often more than one potential design which fulfills all design requirements. In such cases, design trade-off studies are performed to compare the different design alternatives with respect to the design requirements. Design trade-off studies appear during multiple phases in the design process including: (a) during the conceptual design phase and (b) during the detailed design phase. In the conceptual design phase, there are design trade-off studies between different architectures, which require minimal computational resources compared to detailed analysis, because most of the time surrogate models are used for each architecture option. In traditional design processes often a *single* architecture is selected, which eliminates all other architecture options. Then the selected architecture is fully developed to a detailed design. Detailed designs are subject to further design trade-off studies by changing design parameters or features and evaluating robustness properties of the system. Regardless of the parameter variations, the architecture remains intact at this phase of the design process.

MBSE tools can be used to model design alternatives and variations of products. A SysML-based design chain information modeling technique is presented in [157] for variety management in production reconfiguration. They consider a switchgear enclosure production process: a SysML model is developed to perform structural, behavior, constraint, and requirements analysis for reconfiguration of the process. The generic design structure contains: (1) product structure, (2) variety feature, and (3) configuration constraint. A product structure represents a single architecture choice, which is shared by all variants from a product family. The variety features include color, material, and thickness of the switchgear enclosure. The configuration constraint specifies a set of rules that must be held by the final product. One example is the material compatibility: the material of the top, bottom, and vertical brackets must be the same. Another example is the color compatibility: the color of the top, bottom, and vertical brackets must be the same. Even though variety is allowed in the product family, all design requirements and configuration constraints must be met by any design configuration.

If developing or virtual prototyping of multiple design alternatives is unreasonable, for instance due to cost or time, then certain key decisions must be made in advance. Designers need to make decisions regarding technologies, architectures, design solutions, or products to use based on the functional and non-functional design requirements. A method for analyzing requirement trade-offs in the absence of numerical data is presented in [43]. Using the proposed algorithm to generate all consequences of alternatives, the Even Swaps Multi-Criteria Decision Analysis method can be used to guide stakeholders to the best solution. Each requirement is mapped to one or multiple goals and a satisfaction value is defined for each alternative. The satisfaction value is a number between 0 and 5 and is defined by the stakeholders. The goals (G_n), design alternatives (A_m), and satisfaction levels $Sat(G_n, A_m)$ are summarized in a table for each pair of alternatives. Based on the algorithm a series of questions is asked to the stakeholders; for example, if we reduce $Sat(G_4, A_1)$ to 0 from 5, how much improvement would you expect in $Sat(G_1, A_1)$? The goal of the questions is to bring all alternatives to the same satisfaction level for a specific goal, after which that particular goal can be removed from the table. The questions are asked until there is only one goal remaining, probably the most dominant one. Finally, the design alternatives are ordered based on the resulting satisfaction levels.

Simulation-Driven Design

Simulation-Driven Design (SDD) is a design process where design decisions are mainly based on computer-based modeling and simulation of the performance behavior of the designed product in all phases of the design process [134, 135, 139]. SDD is based on Model-Based Development (MBD) with simulation support to allow validation of various architecture options even in the early phases of the design process. One key difference between SDD and MBD is that in SDD certain models or model types must be *executable* by one or more simulation tools, where the execution semantics are well-defined by the simulation tool(s). Therefore, SDD enables designers to make design decisions based on how each decision will impact the performance behavior of the designed product. SDD considers the impact on performance, which helps to reduce the number of prototypes to build, resulting in reduced development and product costs. Using simulation techniques may also increase product quality during the different design phases. This design process often requires a tight interconnection among various tools used during a design process such as product management tools, simulation tools, requirement management tools, etc.

SDD has been adopted by different MBSE tools such as SysML. A Simulation-Driven Design using SysML is presented in [123], where parametric SysML diagrams are instantiated with values; and solutions can be derived by solving the equations of the composed system.

An analysis of a complex system for electrical mobility using a model-based engineering approach focusing on simulation is presented in [151] using a hypothetical electric car (eCar) example. The SysML4Modelica profile is used to model the eCar system, which consists of three heterogeneous domains: software, electrical, and mechanical domains. The question to be answered: is a 1- or a 2-motor eCar concept more advantageous? To answer that question, the SysML4Modelica based eCar model, including all domain models, is translated to a Modelica model to perform the simulation and evaluate both architectures. According to the simulation results the 2-motor concept is $5-7\%$ more efficient than the 1-motor concept. When such a question is asked there always should be a quantifiable evaluation criteria: in this case the requirement was the minimum charge of the battery must be at least 90%. The requirement is also translated to the Modelica model as a state machine, which evaluates the requirement during the simulation and indicates whether the requirement is violated or not.

A system-level model integration of design and simulation is presented for mechatronic systems based on SysML in [26]. Light-weight extensions were added to SysML to represent SysML-based hybrid system dynamic behavior models such as controllers or computation blocks and physical dynamics. One extension is called Sequenced Parametric Diagram (SPD), which is an extension of the SysML parametric diagram to describe hybrid dynamic behavior. The other extension defines stereotypes associated with specific SimScape semantics: e.g., sensor, physical, actuator, hybrid, scope, and simulation. The SysML models are transformed to Simulink Stateflow and SimScape for simulation purposes. The bidirectional transformation between SysML models and Simulink Stateflow and SimScape models is implemented using Triple Graph Grammar (TGG) method [76].

The previous extensions to SysML have been improved and generalized. A SysML-based uniform behavior modeling and automated mapping of design and simulation model for complex mechatronics is developed as a SysML profile called Uniform Behavior Modeling Language (UBML) [25]. UBML unifies modeling concepts from Simulink SimScape and Modelica models and adds them as extensions to SysML. One of the key concepts is the *simulation diagram*, which is used to define a simulation for a design including the simulation parameters (such as start time and stop time) and stimulus to the system. The simulation diagram can be translated to either Simulink Stateflow and SimScape or a Modelica simulation tool called MapleSim [93].

Simulation-Based Design using SysML is also applied in the solid modeling and analysis domain [124]. SysML block diagrams can be linked to Computer-Aided Design (CAD) parts, which can be parameterized in a parametric diagram by defining associations between the different component ports. The parameters are three dimensional positional constraints and sizes such as height, width, and length that are defined between the different geometric parts. Constraints are modeled as SysML constraint blocks with constraining equations between the inputs and outputs. These parametric diagrams are defined for CAD models and Finite Element Analysis (FEA) models. FEA models are used for detailed thermal and structural analysis, e.g., finding the maximum load which the part or assembly can withstand without deformation. The parametric diagrams that parameterize *existing* FEA models are called *analysis templates*.

In addition to SDD and analysis related extensions, failure modeling concepts can also be used in SysML using the Dysfunctional Behavior Database (DBD) extensions for SysML. DBD has been added to support reliability studies of complex physical systems using rapid failure mode identification during the Failure Modes and Effects Analysis (FMEA) process [37]

Multidisciplinary Design Analysis and Optimization

Multidisciplinary Design Analysis and Optimization (MDAO) techniques are used to find the best and optimal solution from all design alternatives, which can be viewed as the second phase of the design trade-off studies [34, 78, 137, 138]. These analysis and optimization techniques are applied after the number of possible design architectures are pruned to a manageable set of alternatives. MDAO automates running the analysis (e.g., simulation) to explore the parametric design spaces given a set of constraints and bounds on the input parameters. A set of input parameters for a given execution is called an input vector. The automation of the analysis processes is critical, because each analysis must be executed multiple times with a different input vector for each iteration. The input vector is defined by the selected analysis or optimization method. For example, the input vector can change material properties, loads, manufacturing tolerances, or operating conditions of the design. The analysis results are collected and presented to the designers to help to make design decisions. These analysis results are also used to understand the robustness of each design, generate a surrogate model, or find an optimal input vector for the design problem (e.g., best material). This method involves iteration over a continuous parametric space, resulting in a high demand on computational resources.

There are several existing MDAO tools (e.g., OpenMDAO [31, 54, 56], ModelCenter [92], Dakota [79], Optimica [4], iSight [145], etc.) that provide the aforementioned capabilities or the subset of them. OpenMDAO [31] is a high-performance computing platform for system analysis and Multidisciplinary Design Analysis and Optimization. It provides a library of solvers and optimizers including design-of-experiment drivers, surrogate model generators, gradient-free optimization methods, and gradient-based optimization methods. The solvers and optimizers can execute the analyses in parallel locally or on a cluster of computers.

An object-oriented and executable SysML framework is presented for rapid model development [11]. This framework uses a web-based technology to support collaborative authoring of SysML models and it utilizes OpenMDAO to facilitate the execution of the SysML models for MDAO problems.

Lessons learned

Creating Model-Based Systems Engineering (MBSE) tools for complex CPS design involves several challenges [146, 148]. As we have seen, existing and current product design processes fail to address the challenges of capturing and evaluating requirements, integrating heterogeneous methods from different disciplines, integrating multiple analysis models with tightly- or loosely-coupled executions. Unfortunately, some of the design processes strictly focus on a single discipline or analysis tool, often on simulation-based analyses. However, other important aspects exist in a complex design project, such as fault modeling and analysis, manufacturing process modeling and analysis, or the use of formal verification methods. This implies that there is no single MBSE tool that can accommodate all aspects of a complex CPS design problem. Each discipline-specific tool must be used within the scope for which it was designed. Many of these tools must communicate through a common platform and share some aspects of their models and tool-specific settings or configurations. Creating analysis definitions for analysis tools often requires subject-matter expertise, but existing analysis definitions can be reused for different design problems from the same application domain. This is possible by bringing tool-specific concepts to a higher level of abstraction, opening the space to a new group of designers who have limited or zero subject-matter expertise. By advocating model reuse across multiple tools, existing labor intensive work is leveraged and development time is reduced. For example, a solid model can have multiple fidelity levels, and can be reused by multiple analysis tools: computing the center of gravity, running a structural or thermal Finite Element Analysis (FEA), or analyzing manufacturability properties.

As we presented in the previous sections, many extensions have been added to SysML. For example, the Uniform Behavior Modeling Language attempts to bring modeling concepts to SysML from Modelica and SimScape to represent behavior of system dynamics. Even though a mapping between Modelica and SimScape concepts is presented in the paper, the SysML model has only some correspondence to the transformed simulation models. The mapping between Modelica and SimScape models is possible, but at a very high level of abstraction. Both Modelica and SimScape provide an object-oriented equation-based modeling language and component libraries for various application domains such as mechanical, electrical, etc. If the mapping is at the component library level with elements like motor, reducer, ball screw, mass, sensor, gain, adder, etc., then it is possible to transform UBML models to both Modelica and SimScape. However, there are two important conditions that must be met:

the structural semantics and the execution semantics must be the same between Modelica and SimScape for the used elements. In addition to mapping the component libraries, the parameters and units of the components must also be mapped. Each tool differs in how they represent more complex domains like multi-body dynamics or fluid. There is *no* general one-to-one mapping between the Modelica language and the SimScape language. For example, in Modelica, there is a `Modelica.Mechanics.MultiBody.World` component that defines gravity, gravity type, gravity field constant; in SimScape, the analog is achieved by connecting the *World Frame* block and the *Mechanism Configuration* block. MATLAB provides Simulink and Stateflow which are rich toolboxes used for controller modeling, whereas Modelica has only minimal support for controller modeling. The preferred industry practice is to use Simulink for controller design.

CHAPTER III

HIGH-LEVEL DESIGN FLOW

This chapter presents an overview of the key concepts of our high-level design flow. Reusable design processes and methodologies should be independent of both the target application domain and of the Model-Based Systems Engineering tool. The presented key concepts are independent from any implementation, but our implementation is referenced where applicable. To facilitate an extensible rapid design environment, four layers are identified as integration platforms to address different concerns in the design process: (1) Model Integration Platform, (2) Tool Integration Platform, (3) Execution Integration Platform, and (4) Visualization Integration Platform. Each integration platform has a set of domain independent and a set of domain dependent aspects. The domain dependent aspects capture the structural or behavioral semantics of each domain, whereas the domain independent aspects capture concepts or utilize tools regardless of which domain is used. Certain domain independent concepts are mapped to multiple domains to provide consistency across domains. For example, the model parameters are defined once and they could parameterize multiple domain models. Another example: a domain model can be reused for multiple analyses such as dynamic simulations and formal verification.

The Model Integration Platform is used to develop system models that comply with semantically rigorous model integration languages and defines the semantic concepts required for multi-domain and multi-disciplinary design. In the OpenMETA tools [69], the Model Integration Platform is primarily facilitated by a model integration language called the Cyber-Physical Modeling Language (CyPhyML). These models are used by the Tool Integration Platform. The Tool Integration Platform provides a set of tools that (a) preserves relevant portions of external models that are required for integration and (b) generates analysis packages for the Execution Integration Platform. These tools work on models defined in the Model Integration Platform and in external model-based engineering tools. The Execution Integration Platform provides a model-based engineering tool-independent way to execute the analysis packages. The executed analysis packages create analysis results in a form that the Visualization Integration Platform can visualize. Designers use the Visualization Integration Platform capabilities to analyze their multi-domain and multi-disciplinary designs and design families and to perform design trade-off studies. Each integration platform, shown in Figure 6, is presented in detail in the following sections.

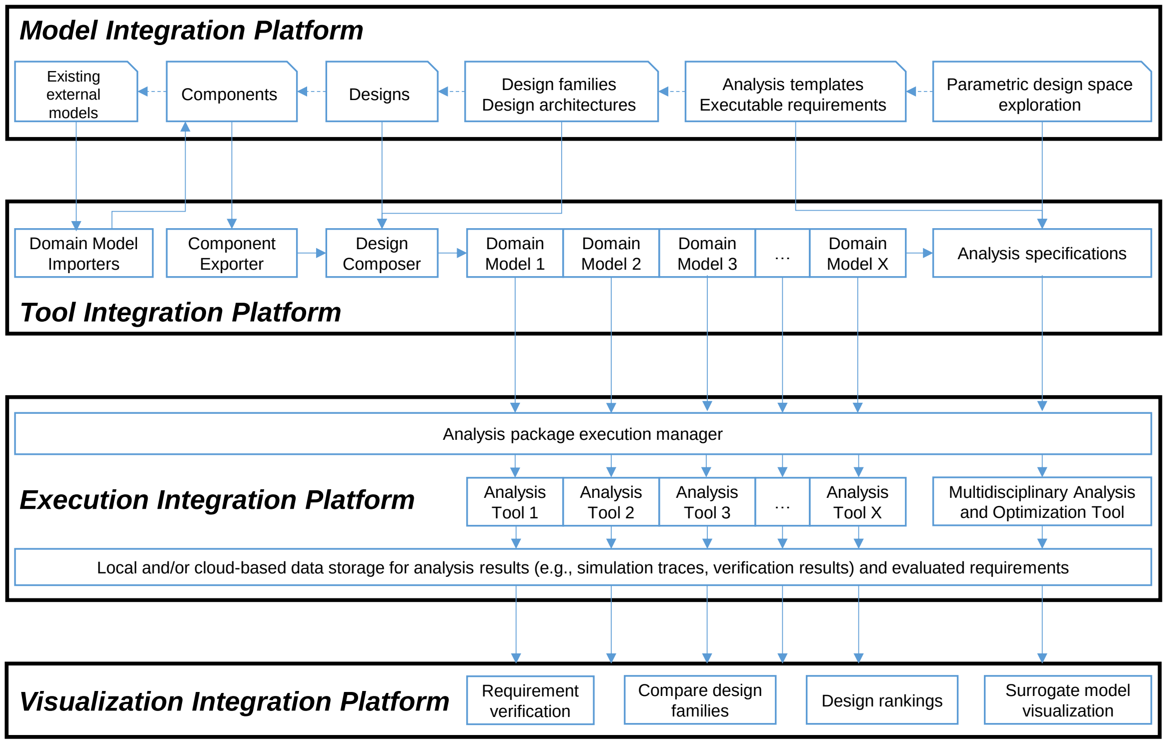

Model Integration Platform
Existing external models
Components
Designs
Design families Design architectures
Analysis templates Executable requirements
Parametric design space exploration
Domain Model Importers
Component Exporter
Design Composer
Domain Model 1
Domain Model 2
Domain Model 3
…
Domain Model X
Analysis specifications
Tool Integration Platform
Analysis package execution manager
Execution Integration Platform
Analysis Tool 1
Analysis Tool 2
Analysis Tool 3
…
Analysis Tool X
Multidisciplinary Analysis and Optimization Tool
Local and/or cloud-based data storage for analysis results (e.g., simulation traces, verification results) and evaluated requirements
Visualization Integration Platform
Requirement verification
Compare design families
Design rankings
Surrogate model visualization

Figure 6: Workflow and integration platforms for the design process

Model Integration Platform

The Model Integration Platform defines the semantic concepts required for multi-domain and multi-disciplinary design. Figure 6 depicts the key concepts, the logical relationships between them, and the interaction between the Tool Integration Platform software components. The key concepts are: (1) external models, (2) component models, (3) design models, (4) design family models or design architecture models, (5) analysis templates or executable requirement models, and (6) parametric design space exploration models. The Tool Integration Platform is used to import existing domain-specific models as external models. External models capture information about interfaces required for composition of various domain models. External models from different domains are kept inside component models and the external model interfaces are mapped to the component model interfaces to enable composition of component models. By composing component models we form a design model that provides a full representation of the system within its boundaries. A single design model is often called a single point design or seed design. The design family models represent multiple design models, often called a discrete design space, where the selection of component models and architecture variations are not yet fixed. Therefore, a single design model is a subset of a design family model. Because the design and design family models do not encapsulate the environment of the system, often they cannot be analyzed or evaluated against a set of requirements unless they are put into a context. *Analysis templates* or executable requirement models define the context for a system in which it can be evaluated. The analysis templates are directly linked to the requirements of the system or subsystem and are tied to a domain-specific analysis tool that performs the evaluation of the system model. If an analysis template is built for a design model, it becomes reusable on design family models as long as the design family model has identical interfaces with the design model. This approach allows designers to reuse their models and analysis setup over several design points. The analysis templates have an interface with input parameters and generated outputs. Parametric design space exploration models are defined to further explore the overall design space. These models contain a driver model which represents an optimizer, a Design of Experiment (DoE) study, or a *Probabilistic Certificate of Correctness (PCC)* [61] study. A driver model selects the method used to explore the parametric design space by generating a set of input parameters for an existing analysis template model. A PCC driver generates the input parameters based on probability density functions. These input parameters and their ranges are specified in the driver model, and associated with one or multiple environment, system, subsystem, or component parameters.

Tool Integration Platform

The Tool Integration Platform provides a set of translations between models defined in the Model Integration Platform and in external model-based engineering tools. These translations either extract relevant information from models built in model-based engineering tools and relate that information with models in the Model Integration Platform or the translations (model transformations) generate *analysis packages* for the Execution Integration Platform. Figure 6 depicts the key functions, logical relationships between them, and the interaction between the Tool Integration Platform and the other two platforms (i.e., Model Integration Platform and Tool Integration Platform). The key functions are: (1) domain model importers, (2) component model exporter, (3) design composer, (4) domain model exporters, and (5) analysis specification exporter.

Domain model importers read existing domain models (e.g., Modelica or Computer-Aided Design (CAD) models), import relevant information to the Model Integration Platform in the form of external models, and associate them with component models.

The component model exporter exports the component models to an interchangeable format (e.g., AVM Component model [104, 105]) to represent all domain models and component-level interfaces including parameters and properties of the component.

The design composer exports design and design family structures into an interchangeable format (e.g., AVM Design model [104]) that contains information about components, subsystems, and the composition of component models and subsystems. The Design Composer discovers all component models within designs and design families and invokes the Component Exporter for each included component model.

For each domain a separate domain model exporter is implemented or an existing domain model exporter is reused if the analysis tool requires similar artifacts from the analysis templates. For example, formal verification tools can take Modelica models as inputs along with information about what properties must be verified. If a Modelica model exporter already exists, then the formal verification tools can leverage the output of that exporter. As another example, Computational Fluid Dynamics (CFD) or Finite Element Analysis (FEA) tools require the composed geometric model to perform the analysis. If a CAD model exporter already exists, then the CFD or FEA tools can leverage the output of that exporter. Therefore, these domain model exporters should be created in a way that they can reuse each other's functionality.

In order to execute the generated models an analysis specification must define the execution steps. An analysis specification is always exported for each analysis template and for each parametric design space exploration model. The exported composed domain models along with the analysis specifications form an analysis package. This analysis package is provided to the Execution Integration Platform for execution. The use of the Model Integration Platform and analysis templates makes it possible to automatically generate analysis packages for entire design families.

Execution Integration Platform

The Execution Integration Platform provides a model-based engineering tool-independent way to execute the analysis packages. Figure 6 depicts the key functions, logical relationships between them, and the interaction between the Execution Integration Platform and the other two platforms (i.e., Tool Integration Platform and Visualization Integration Platform). The key functions are: (1) analysis package execution manager, (2) analysis tools, (3) Multidisciplinary Design Analysis and Optimization (MDAO) tool, and (4) local or cloud-based data storage for analysis results.

The analysis package execution manager is a batch job executor that takes analysis packages as its inputs and schedules individual jobs to execute them. The jobs are executed either locally or remotely depending on how the execution manager is configured.

The analysis packages may require one or more analysis tools installed on the remote machines; the execution manager must verify that the remote machine has the appropriate tools. The analysis tool and platform requirements are set when the analysis packages are posted to the execution manager. After the execution is finished, with either a success or a failure, the analysis results are retrieved for the user. For remotely executed analysis jobs the results are saved in a cloud-based data storage.

The MDAO tool could be treated as another analysis tool, but often times it requires more computational resources than are available in a single computer. MDAO tools are computational intensive tasks, when the analysis tools are executed several hundred or thousand times. Because each analysis tool has a different result format, post processing functions are created to map the raw data to system or subsystem level requirements. The final results are stored in the structured analysis specification document. All the generated results are available for the Visualization Integration Platform for further evaluation.

Visualization Integration Platform

The Visualization Integration Platform takes a set of executed analysis packages from the Execution Integration Platform along with their results and provides several visualization techniques to guide the designers to improve their designs and make design decisions. In the OpenMETA tools, the Visualization Integration Platform is implemented by the Project Analyzer. Figure 6 depicts the key visualization techniques: (1) requirement analysis, (2) comparing designs and design families, (3) design ranking, and (4) surrogate model visualization with prediction profiler and 3D constraint plots.

Each design problem defines a set of requirements and these requirements are often defined in a textual form. The textual requirements are (manually) translated to a structured form that allows mapping of requirement values to analysis template results. The Visualization Integration Platform provides requirement verification tools to visualize the evaluated requirements for individual design candidates and for entire design families. For instance, the parallel axis plot is a common way to analyze multivariate data and visualize high-dimensional geometry (examples are shown in Chapter VI along with two use cases).

If two design candidates meet all the requirements, additional visualization tools may be required to further compare these candidates. For each design or pair of designs the component selection or architecture choices are visualized in a table. Another visualization technique shows clusters in the overall design space based on selected components. The numerical outputs of the design candidates are shown on parallel axis plots or a comparison table. When there are more than a few outputs, it becomes challenging to pick which design is the best. Some designs perform better in one aspect, others perform better in other aspects.

The Visualization Integration Platform uses the Technique for Order of Preference by Similarity to Ideal Solution (TOPSIS) analysis to create a relative ranking of the designs. TOPSIS is a multi-criteria decision analysis method [62, 63, 158], which is used to rank designs based on the geometric distance from the positive ideal solution and the negative ideal solution. This way each design candidate gets a single numerical value between 0 and 1 based on the user specified weights on the analysis outputs, where 1 means the most ideal solution and 0 means the least preferred solution. Because a single value is assigned to each design a simple ordered list of designs can be produced to guide the designers to pick the best design candidate(s).

When a few possible design candidates are identified by using the design ranking, further design analysis might be required. Even if a design meets all requirements under normal operating conditions, it can be sensitive to changes in the environment or on uncertainty of the component properties (e.g., manufacturing tolerances). Therefore, MDAO tools are used to evaluate the parametric design space that captures such variations. The MDAO tools can produce a surrogate model, which serves two purposes: (a) analyze the sensitivity of the design around the optimal operational conditions and (b) execute the surrogate model instead of the analysis within the valid parametric space. The Visualization Integration Platform visualizes the generated surrogate models using a prediction profiler or as 3D constraint plots. Within the valid parametric design space the analysis outputs can be predicted with a given set of inputs without executing the analysis.

CHAPTER IV

Heterogeneous component models

Problem Statement

The goal of Model-Based Systems Engineering (MBSE) is to improve the efficiency and the quality of the design process, which can be partially achieved by: (a) using heterogeneous component models, (b) keeping the multiple domain models consistent, (c) establishing and tracking model dependencies, and (d) importing existing model libraries into MBSE tools.

Creating well-defined component interfaces and capturing multiple domain models (views, concerns) for each component helps to identify and track cross-domain interactions. Using component interfaces provides component- and subsystem-level reusability and continuous refinement of the design within an iterative design process.

Domain models can represent multiple views of a component such as behavioral, structural, or manufacturing (e.g., the lead time or the required manufacturing process). Components capture and encapsulate the dependencies between those views. Consider a parametric gear component where the radius of the gear can be changed within a range. When the gear radius changes, it *must* affect the behavioral, the structural, and the manufacturing view of the component. Depending on the gear radius: (a) the dynamic behavior of the system changes, (b) the gear claims more or less space in 3D and affects the location of the adjacent components and the overall weight of the design, and (c) the gear may require a different manufacturing process.

Developing model libraries is a costly, labor intensive, and time consuming process. Therefore, designers prefer to use their existing model libraries for designing new systems. A few new models might be developed for the new system, but most of the models are reused. Several analysis tools provide libraries of components as part of their software distribution. In addition to the built-in libraries, open-source and commercial packages are developed for those domain-specific tools. Designers prefer to develop, test, debug, and validate their libraries using the domain-specific tools.

When a new Model-Based Systems Engineering tool is used, it is important to consider whether existing model libraries can be used or if designers should create new libraries from scratch. To ease the friction in tool adoption, the following question must be addressed: how to facilitate the mapping between existing model libraries and the Model Integration

Platform for both *causal* and *acausal* models. The model libraries can contain a set of components (i.e., building blocks), designs, or test cases for the components.

Challenges

A wide variety of Model-Based Systems Engineering (MBSE) tools are used in product designs and frequently models are created in distinct tools. These models are often organized into model libraries, which leads to a research challenge: how to facilitate the mapping between existing model libraries and a Model-Based Systems Engineering tool for both causal and acausal models?

After the aforementioned mapping is established another research challenge arises: how to define reuseable multi-aspect and multi-view component and design models, where component models and design models can be reused for designing product families?

Consider dynamic simulation models implemented using different acausal modeling languages: e.g., Bond Graph and Modelica. When libraries of components are developed in both languages, it is more convenient to use the libraries as-is rather than migrating an entire library of components to another modeling language. Our goal is to develop a modeling language that supports heterogeneous dynamics model composition that addresses how to reuse existing Bond Graph and Modelica models within the same modeling environment.

Component and design models

Solution

Our solution described below contributes to the Tool Integration Platform and implements concepts from the Model Integration Platform including: (a) referencing existing external models, (b) component models, (c) design models, and (d) design family models. The full specification of our implementation is represented by a meta-model called the Cyber-Physical Modeling Language (CyPhyML); the formal semantics are defined by using the *FORMULA* framework [71, 72] and presented in [105, 136]. In this section we present the key concepts of our solution as a UML class diagram shown in Figure 7. We use the UML class diagrams to illustrate the key concepts, relationships, and attributes.

CyPhyML is a model integration language that helps to integrate models from external tools. Figure 8 shows that the component models encapsulate domain models that refer to *existing external models* built in external tools. These existing external models may represent physical entities, software, or a cyber-physical component that contains both. The domain models refer to an external resource (the model contained in another tool), and capture the

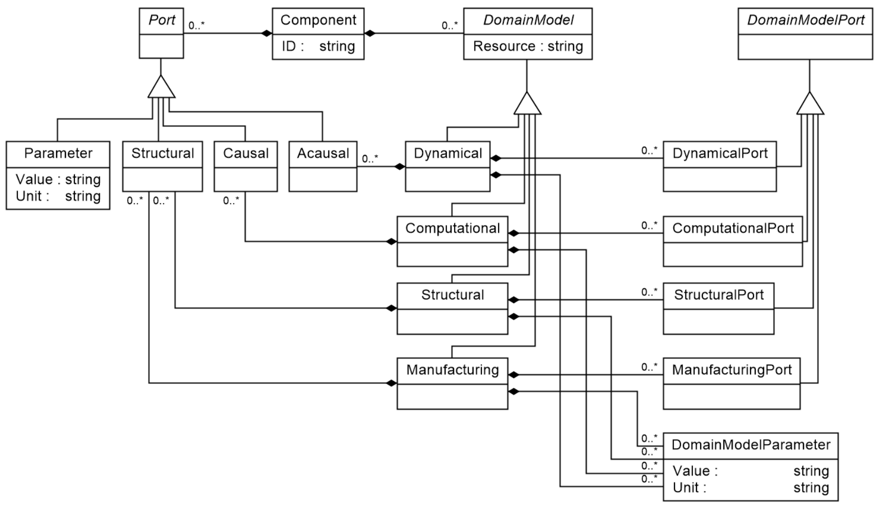


Figure 7: Component class diagram

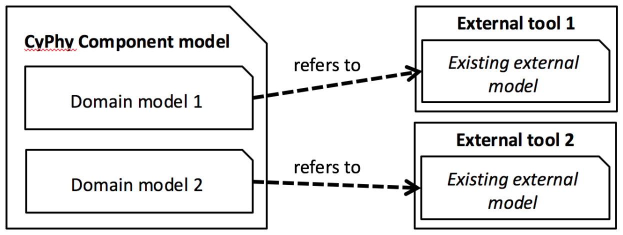


Figure 8: CyPhy Component model and existing external models

interfaces of that model, which are required for model composition. The implementations of the existing external models are *not* replicated in these domain models, but rather the domain models link and track the external model dependencies. In CyPhyML, there are four different domains: (1) dynamical (e.g., Modelica, Bond Graph, or SimScape), (2) computational (e.g., Simulink StateFlow), (3) structural (e.g., PTC Creo part or assembly), and (4) manufacturing (e.g., lead time of the component, required manufacturing process). Each domain model has a set of interfaces that include domain-specific ports and domain model parameters. The domain-specific interfaces are mapped to the generic component-level interfaces called ports. Some interfaces are causal (*Parameter* and *Causal*) and some are acausal (*Structural* and *Acausal*). In this chapter we present an example for model integration, where the causal and acausal interfaces and their interactions are explained in detail.

Component models represent Cyber-Physical System (CPS) components that capture and encapsulate one or multiple domain models. Domain model interfaces are extracted from external models by domain model importers. A domain model importer builds an external model within a CyPhy Component model and it is part of the Tool Integration Platform. The extracted interface information and the external domain model are associated

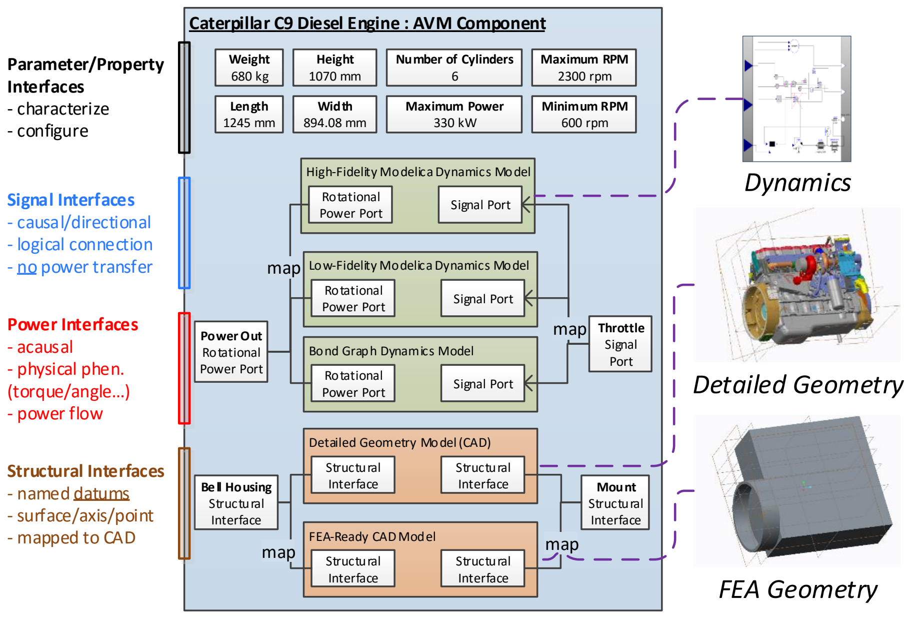


Figure 9: AVM Component Model for Caterpillar C9 Diesel Engine [105]

with these component models. Each component model has a set of causal and acausal interfaces that participate in the composition. Component-level parameters are used to keep the independent domain model parameters consistent by direct connections between the component-level and the domain-level parameters. In CyPhyML, a component model is called a *CyPhy Component*. CyPhy Component models are packaged with all domain models and a component descriptor, which forms an *AVM Component* model. Figure 9 gives a conceptual view of an example of an AVM Component for a Caterpillar C9 diesel engine. This example highlights the four interface types and contains: (a) a set of component parameters (e.g., weight, height, maximum power), (b) three different dynamics models (a high-fidelity Modelica model, a low-fidelity Modelica model, and a Bond Graph model), and (c) two CAD models (a detailed geometry and a simplified CAD model for FEA). The composition of several component models forms a design model.

Figure 10 shows the design model and the discrete design space model as a UML class diagram. A design model is called a *Configuration* in the UML class diagram; it is called a *CyPhy Component Assembly* in CyPhyML. A Configuration defines a set of components and their interconnections, which is the full representation of a system. A set of Configurations is represented as a discrete design space model, which can be used to model product families. There are two types of design spaces: (1) *parametric design space* and (2) *discrete design space*. The parametric design space is discussed in detail in Chapter V. The discrete design

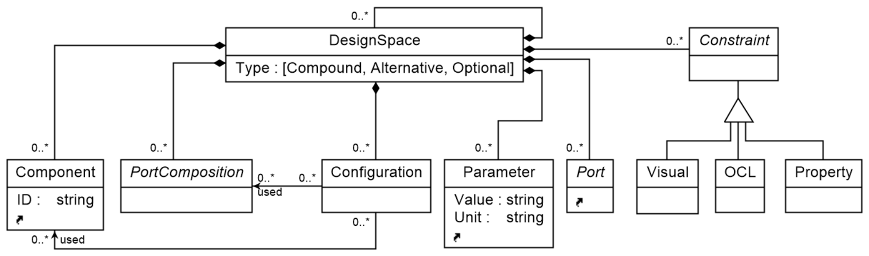


Figure 10: Design space class diagram

space represents finite number of designs (i.e., configurations) and it is often called a combinatorial design space [110]. All possible configurations can be generated by enumerating over the choices in the design space. This discrete design space contains two kinds of choices: (1) component choices and (2) alternative topologies (i.e., variation of interconnections between components). The discrete design space is recursive and hierarchical. In other words, design spaces can contain other design spaces. In the OpenMETA tools, the discrete design space and its tool support are facilitated by the Design Space Exploration and Refinement Tool (DESERT) [23, 107, 108, 110, 111, 156].

Figure 10 shows the key discrete design space concepts. Each design space has a type; it is either compound (i.e., mandatory model), alternative (i.e., select exactly one component or model from a valid set), or optional (i.e., select at most one component or model). By adding more and more component choices to alternative or optional containers, the size of the design space grows quickly. To keep the design space to a manageable size, three types of constraints [110] can be added: (a) visual constraints (e.g., to express symmetry in the design), (b) constraints expressed using the Object Constraint Language (OCL), and (c) property constraints (e.g., cumulative cost of components cannot exceed a certain value). Applying these constraints will eliminate configurations that would violate the constraints.

The interfaces of the design space models are identical to the interfaces of the component models. This provides continuous design refinement, where components can be elaborated into subsystems and design spaces. As a result, component and design space models are reusable and interchangeable. The design space models generate many configurations, where each configuration is a full specification of a design with a selected set of component choices, interactions, and topology.

Evaluation

In this chapter we presented our contribution to the Model Integration Platform and Tool Integration Platform that addresses the challenge of creating heterogeneous component models. Our solution is extensible for other domain models (e.g., circuit design) and has been demonstrated in extensions performed by other researchers using the OpenMETA tools [99]. The domain extensions will affect the component model specification, but the design space models and tools remain intact. Our approach is applicable if there is an existing library of composable models and the hierarchical decomposition of the designed system is identical for all domains. In the following section, we present a detailed example for model integration that considers causal and acausal interfaces and multiple modeling languages such as Modelica and Bond Graphs.

Example for model integration

In order to describe the composition of CPS, we must clarify the concept of causality. Component models are composed by connections between their interfaces (i.e., ports). These connections represent relationships between the connected interfaces. This relationship is either a *causal* or an *acausal* relationship. The relationship is considered causal if it is a *cause-effect* (i.e., $y := x$) relationship between the interfaces. In other words, the value of x always determines the value of y, but y does not determine the value of x. The relationship is considered acausal if there is a *constraint* (i.e., $x = y$ or $x + y = 0$) relationship between the interfaces. In other words, for all instants of time, either x is the same as y, or the sum of x and y is 0, but there is no causal relationship defined. A more precise definition can be found in [153]. Typically, computational systems are causal, while physical systems are acausal.

Using causal models (e.g., signal data flows) to represent interactions between components that share physical variables can be complex. Typically, acausal physics models have power ports, which represent a simultaneous, bidirectional energy exchange between components [80, 118]. A well-formed model in an acausal framework represents a well-formed set of dynamic equations. Acausal models typically must interface with causal models to represent the integration of a controller function into a physical system. This requires carefully directed variable sharing between cyber and physical system components (e.g., through sensors and actuators).

Our solution is presented in the context of a simple electro-mechanical system. Some dynamic system components are created using Bond Graphs, others are created using Modelica. Both types of components are used in the integrated system model that describes the system behavior. The component models are connected through ports using acausal connections. Because the components can have either Bond Graph or Modelica models, a tool has to be selected to execute the integrated system model. We selected the Modelica language and a Modelica tool as our simulation execution environment.

Solution

We present the syntax and semantics of such an integration language and its component-based design, where components can embed models from different tools, formalisms, and paradigms such as Bond Graphs and Modelica models. Our framework is built around a common set of interface concepts to support heterogeneous composition and interchangeability among modeling paradigms.

In order to drive our focus, we needed a component-based modeling framework of such systems, where component models were coming from different tools. We illustrate this through a case study using a simple electro-mechanical system. A schematic of the system is shown in Figure 11. The system is modeled using Modelica models and the simulation can be executed using a Modelica tool. The example uses three different domains: (i) electrical ($stepVoltage$, $resistor$, $inductor$ and emf), (ii) mechanical rotational (emf, $inertia$, $damper$ and $idealRollingWheel$), and (iii) mechanical translational ($idealRollingWheel$ and $mass$).

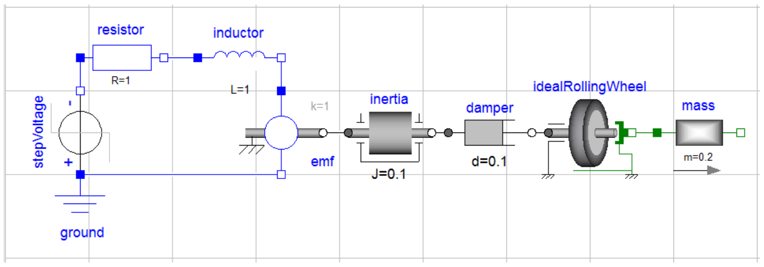


Figure 11: Schematic diagram of a simple electro-mechanical system [84]

To model further components of the same system (e.g. $resistor$ and $idealRollingWheel$), we would like to use the formalisms of another modeling paradigm, namely Bond Graphs. In order to support this setting in a design environment, we need the following: (i) a common, consistent modeling framework that can interface to models that are based on various formalisms and paradigms, (ii) a composition approach that is able to integrate and simulate the system as a whole, and (iii) the ability to adapt the system models to widely used tools in order to able to simulate the composed system.

In the following, we illustrate the objective with an example for the expected solution. Regarding the case study, this means that we create the same system along with its connections shown in Figure 12. The puzzle pieces represent components, which are Modelica models. The expected solution along with its simulation results is shown in Figure 12. The plot shows the simulation results of: (1) the angular velocity of the $inertia$, (2) the force on the translational interface of the $idealRollingWheel$, and (3) the current on the positive electrical pin of the $resistor$.

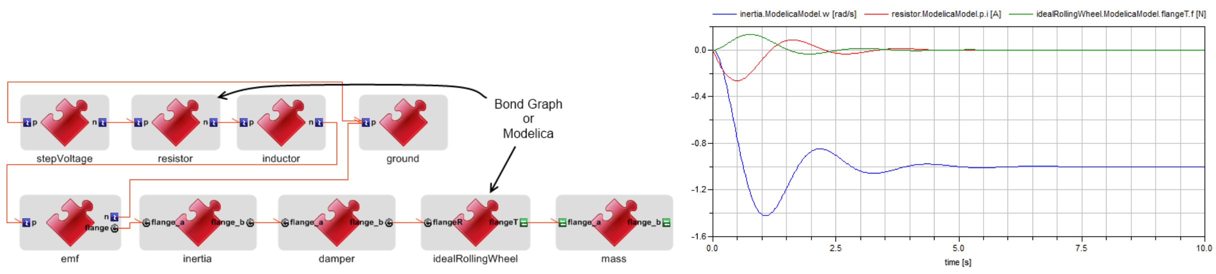


Figure 12: Composition diagram and simulation results [84]

We have changed the underlying model types from Modelica to Bond Graphs for the *resistor* and the *idealRollingWheel* components. The generated simulation results are identical with respect to the variables mentioned above and as it is shown in Figure 12. We used a simple example, where the models of the components are easy to develop either using Modelica or Bond Graphs. A more practical example is a drive train model, where the engine and transmission components can be modeled using Modelica, and the final drive and the load components can be modeled as Bond Graphs. The integration language supports heterogeneous composition of components implemented using Bond Graphs and Modelica. Thus, it should characterize interfaces in terms of the commonalities between the supported modeling paradigms. We will present the integration language and its semantics.

We chose a Domain-Specific Modeling Language (DSML) to implement the integration language. DSMLs are flexible to support iterative refinement and can capture multiple paradigms as well as complex electro-mechanical system domains. We used the Generic Modeling Environment (GME) [68], a meta-programmable editor, for creating this domain-specific integration language [68]. For the Adaptive Vehicle Make (AVM) program [44], a Cyber-Physical Modeling Language (CyPhyML) was created for a diverse set of design tasks. This section describes the subset of an early version of the CyPhyML meta-model (language) and explains in detail the concepts relevant to the behavior of dynamic systems.

CyPhyML has different modeling aspects and one of them is the dynamics aspect of components. In CyPhyML components different behavior models can provide a common set of interfaces, and can be composed through them. Also, the modeling framework supports hierarchical composition of reusable CyPhyML components.

A *Component* in CyPhyML is an atomic building block. A CyPhyML component is defined by its interfaces: (1) parameters, (2) signal ports, (3) power ports, and (4) structural ports. Because the ports are strongly typed, connections are only allowed between port pairs of the same type. This rule, implemented by the modeling environment, forces the designers to build such models that always comply with the structural semantics of CyPhyML. Connections defined for parameters and signal ports always represent causal (i.e., cause-effect) relationships, whereas connections defined for power ports and structural ports always represent an acausal (i.e., constraint) relationship. Each CyPhyML component includes a *behavior model*, a mathematical abstraction of a real-world physical system which represents the component's dynamic behavior. A behavior model can be specified using the Hybrid Bond Graph Language (HBGL) or Modelica. HBGL models can be developed within a component using CyPhyML, while Modelica models are incorporated by referring to an external Modelica model. These references in CyPhyML replicate the parameters, signal ports, and power ports of Modelica models, but do not include the internals or Modelica code. These external models may come from the Modelica Standard Library or from a user-defined library. All domain model interfaces should be mapped to the CyPhyML

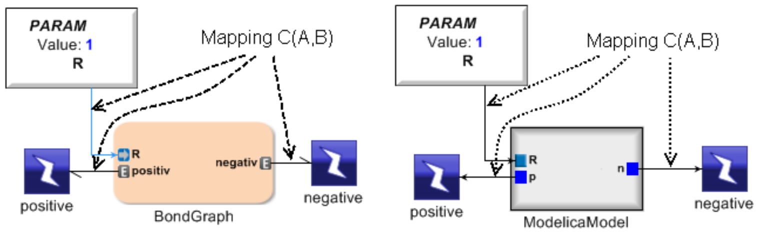


Figure 13: Component model using Bond Graph or Modelica [84]

component interfaces shown in Figure 13. Figure 13 depicts two components representing the same behavior: the component on the left uses HBGL and the component on the right uses Modelica.

CyPhyML components use their own interface definition to support composition between components and to provide common structure among different modeling paradigms. Figure 13 shows an example for the mapping between the CyPhyML component-level interfaces and domain model interfaces. The mapping is defined by a connection $C(A, B)$, where C represents the connection; A is the source element; and B is the destination element. The mapping and the connections, which are described below are only the internal connections that wire the component up to its interfaces. Examples are given for Figure 13. The semantics of the connections are as follows. For parameters (real numbers), $C(A, B)$ means $B := A$ (e.g., Eq. 1) during the model instantiation. This mapping represents a causal assignment between A and B, where A drives B. For signal ports, $C(A, B)$ means the signal value of B is equal to the signal value of A during the entire simulation. It is also a causal connection and A drives B.

$$\begin{aligned} BondGraph.R &:= R \\ ModelicaModel.R &:= R \end{aligned} \tag{1}$$

While power ports are acausal connections and they contain multiple variables at the same time, the mapping needs to be defined through equations using variables of the power ports. A Modelica electrical pin maps to a CyPhy electrical power port: $C(A, B)$ is going to be resolved as Eq. 2 (e.g. Eq. 3).

$$\begin{aligned} A.voltage &= B.voltage \\ A.current + B.current &= 0 \end{aligned} \tag{2}$$

$$\begin{aligned} ModelicaModel.p.v &= positive.voltage \\ ModelicaModel.p.i + positive.current &= 0 \end{aligned} \tag{3}$$

A Bond Graph electrical port maps to a CyPhy electrical power port: $C(A, B)$ is going to be resolved as shown in Eq. 4 (e.g. Eq. 5).

$$\begin{aligned} A.effort &= B.voltage \\ A.flow + B.current &= 0 \end{aligned} \tag{4}$$

$$\begin{aligned} BondGraph.positive.effort &= positive.voltage \\ BondGraph.positive.flow + positive.current &= 0 \end{aligned} \tag{5}$$

A Modelica mechanical translational flange maps to a CyPhy translational power port: $C(A, B)$ is going to be resolved as shown in Eq. 6.

$$\begin{aligned} A.position &= B.position \\ A.force + B.force &= 0 \end{aligned} \tag{6}$$

Bond graph mechanical translational port maps to CyPhy translational power port: $C(A, B)$ is going to be resolved as shown in Eq. 7.

$$\begin{aligned} A.flow &= der(B.position) \\ A.effort + B.force &= 0 \end{aligned} \tag{7}$$

A Modelica mechanical rotational flange maps to a CyPhy rotational power port: $C(A, B)$ is going to be resolved as shown in Eq. 8.

$$\begin{aligned} A.angle &= B.angle \\ A.torque + B.torque &= 0 \end{aligned} \tag{8}$$

A Bond Graph mechanical rotational port maps to a CyPhy rotational power port: $C(A, B)$ is going to be resolved as shown in Eq. 9.

$$\begin{aligned} A.\mathit{flow} &= \mathit{der}(B.\mathit{angle}) \\ A.\mathit{effort} + B.\mathit{torque} &= 0 \end{aligned} \tag{9}$$

In this section we presented an example for model integration. We considered using Bond Graph and Modelica models from existing model libraries and developed a design model for a simple electro-mechanical system. An evaluation of our approach and its possible extensions are provided in the section below.

Evaluation

We chose to use Modelica, which supports acausal modeling, open-source libraries, and has some open-source tools/solvers which can execute the simulations of the composed system models. These tools support optimization techniques and solve the initialization problem for the dependent state variables. The modeling approach described below supports acausal system capture via Modelica Standard Library power ports, with Modelica Standard Library semantics as well as Bond Graphs.

One of the target languages of CyPhyML is Modelica, which means that a translator can generate equations or instances of library elements that use equations (e.g. a Bond Graph library) as well as connect statements for connections. The translator uses interfaces from Modelica Standard Library 3.2 and elements from the Bond Graph library for Modelica if bond graphs are present in the model. Modelica does not have any notion of CyPhyML elements, and the translator works from a semantically rich domain to a semantically poor domain.

After the CyPhyML models have been translated into Modelica models, the hierarchical structure of the generated models resembles the original CyPhyML model's hierarchy. This helps users to navigate their model the same way they do in CyPhyML. The variable tree structure of simulation results look the same, and it is easy to find the appropriate subsystems and plot variables using any Modelica tool.

CyPhyML uses Test Benches to define a simulation or test case for a system. In general, a system itself is not enough to perform a simulation; it requires a context, which contains test drivers and/or environments that interact with the system. Test Benches are described in [85] and are out of the scope of this paper, but were used to generate the simulation results for the examples.

Our approach is not limited to Modelica; because Bond Graphs are the full representation of the dynamics behavior of the components, a set of Ordinary Differential Equations (ODEs) can be derived for each component. For instance, SimScape supports equation-based component models, therefore in our modeling environment one could use SimScape and Simulink models from MATLAB in combination with Bond Graph component models. In this case Bond Graph elements and the composed system model must be converted to SimScape equations and constructs to represent the system behavior.

CHAPTER V

Analysis templates and model execution framework

Problem Statement

Design requirements are often defined in a textual form in the early stages of any design process. Mapping requirements to executable model representations makes requirement evaluation possible. The evaluation of the requirements for complex Cyber-Physical Systems (CPSs) often involves multiple domain-specific analysis tools and analysis types. The analysis types may include dynamic system analyses (including simulations), manufacturability analyses [48], structural analyses, reliability analyses [39, 60, 132], probabilistic verification [61, 103], or formal verification [39, 77, 86]. The different analyses could be interdependent, where the results of one analysis are required as inputs for another analysis.

Consider a family of designs: the same set of requirements must be evaluated. A family of designs may have component variations or architecture variations. These design variations can be explored and realized by combinatorial design space exploration using static constraints. When the design variations are identified the requirements must be evaluated for each design.

As computational resources become more and more available for engineers, support for executing analyses on cloud-based platforms should also be utilized. Depending on the complexity of the design and on the analysis type the appropriate execution platform can be selected.

Challenges

Evaluation of requirements involves the use of one or more analysis tools. A technical challenge is how to design the analysis templates in a way that they can be reused for design families and in a parametric design space exploration and optimization without *any* modification.

Consider a family of designs: the same set of requirements must be evaluated. A family of designs may have component variations or architecture variations. These design variations can be explored by combinatorial design space exploration and pruned by using static constraints. When the design variations are identified the requirements must be evaluated for each design. This presents a research challenge: how to define the analysis templates such that the templates developed for a single design are also applicable and reusable for a family of designs?

The Model Integration Platform defines the modeling concepts and their relationships. Two types of models are executable, but they must first be translated to an executable form for the analysis tools. The Tool Integration Platform contains a set of domain model composers for each aspect of the model. It also includes a generic model transformation tool, which translates analysis templates and parametric design space exploration models to executable analysis packages. To create these analysis packages there are two challenges to solve: (a) how to arrange the analysis packages to facilitate both local and cloud-based analysis execution using heterogeneous analysis tools; and (b) how to keep the concepts and implementation extensible to accommodate custom analysis tools in the future?

Solution

In this section we present our solution for: (a) analysis templates, (b) parametric exploration models, (c) tool integration and analysis package execution, and (d) project artifacts and analysis results management. Our solution described below contributes to the partial or full implementation of all four platforms: (1) the Model Integration Platform, (2) the Tool Integration Platform, (3) the Execution Integration Platform, and (4) the Visualization Integration Platform.

Analysis template models

The Model Integration Platform defines *analysis template models* (i.e., analysis templates or executable requirements) that are executable models used to evaluate one or more requirements. In our OpenMETA implementation the analysis templates are called *CyPhy Test Benches*. The UML class diagram for analysis template models is shown in Figure 14. Each analysis template model contains a *top level system under test* model that refers to a design model or design space model subject to test. Because the design and design space models can have the same interfaces, they are interchangeable. In other words, an analysis template built for a single configuration (i.e., design model) can be reused as-is for an entire design space; and the configuration substitution and execution is fully automated. This design model must be put into a context in which the system is evaluated. The context is provided by test driver models, which represent stimuli to, loads for, or the environment of the system. Sometimes the test subject is a single component of the system. For instance, a structural FEA is only performed on certain key components. In such cases, component references are defined in the analysis template that refer to components within the designed system model.

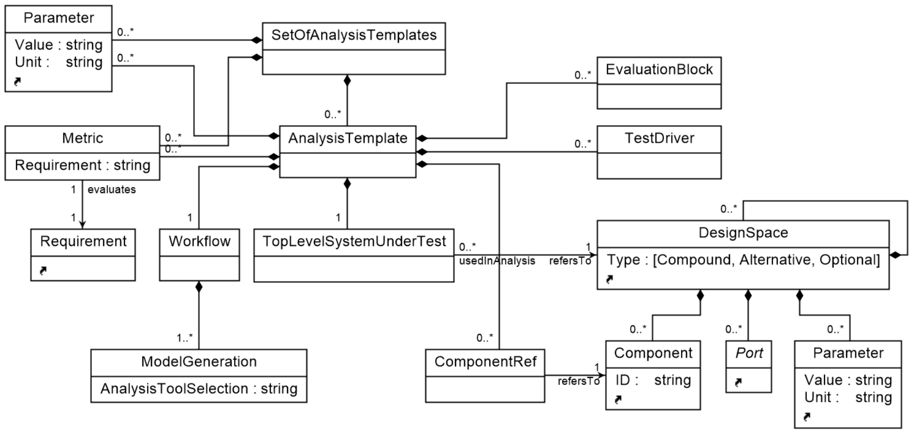


Figure 14: Analysis template class diagram

Each analysis template defines a workflow that contains one or more model generation tools (i.e., model transformations), which are used to create an *analysis package* from the analysis template model. These model generation or transformation tools are implemented in the Tool Integration Platform. The analysis package is an executable domain-specific model for an external engineering tool (e.g., Modelica, Simulink, CFD, FEA, HybridSal).

Each analysis template defines a set of inputs (parameters) and outputs (metrics). The inputs of the analysis template can change the parameters of context model, system, subsystem, or components. The outputs of the analysis template are directly associated with requirements, which makes it possible to evaluate requirements. Parameters and metrics are the interfaces of the analysis templates. The *set of analysis templates* model is an acyclic directed graph for the analysis templates, which are used to group analysis templates. This concept is specifically required when some domain analyses are dependent on other ones. For instance, a dynamic simulation model (e.g., Modelica) requires the mass of the design, which can be evaluated by another analysis on the 3D geometric model (e.g., CAD model). The metrics from one analysis can directly parameterize other analyses. Independent analysis templates are often grouped based on requirement groups, hard and soft requirements, or the involved computational resources.

Parametric exploration models

Parametric design space exploration is a process where the space of parameters of the system is sampled and the corresponding system behavior is analyzed for each sample point. Parametric exploration models are executable models that facilitate parametric design space exploration. A UML class diagram is shown in Figure 15. The parametric design space

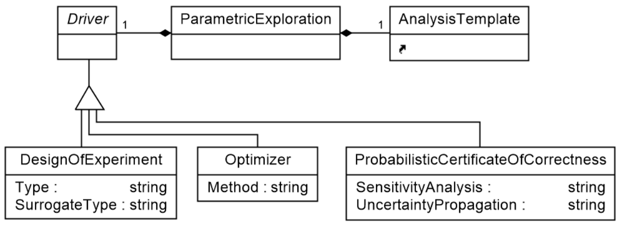


Figure 15: Parametric exploration class diagram

exploration is performed on the contained analysis template model. The analysis templates are used as-is without any changes and they can refer to a discrete design space.

Each parametric exploration model contains a driver, which is one of the following: (a) a Design of Experiment (DoE) driver, (b) an optimizer driver, or (c) a Probabilistic Certificate of Correctness (PCC) driver. Each driver generates a set of input values for the analysis template as parameters according to an algorithm, which depends on the driver type and the selected method. The outputs (i.e., metrics) of the analysis template are constrained, observed, analyzed, and recorded by the driver models. The specification of the DoE driver and optimizer driver types and methods are defined in the OpenMDAO (version 0.8.1 [32]) documentation. The PCC driver is custom driver implemented as an OpenMDAO driver component.

The DoE driver is used to explore the parametric design space within a specified range. It generates multiple input parameter sets for the analysis template and records the outputs. The DoE driver can be configured to create a *surrogate model* for the analysis template. A surrogate model is a simplified model that substitutes the system model with a representation of the system behavior in a valid parametric range. A surrogate model is often used when a detailed model exists but its execution takes excessive time. This surrogate model is packaged with the analysis results in the analysis package and visualized by the Visualization Integration Platform.

The optimizer driver executes a multi-objective optimization using the analysis template with a set of design variables and objectives. The optimizer driver model defines the design variables and their ranges in which the optimization takes place. These design variables define which parameters are used from the analysis templates. The objectives define *which* metrics are maximized and *which* ones are minimized during the multi-objective optimization.

Designing complex CPSs requires a design process that supports maximizing the probability of meeting the design requirements. The Probabilistic Certificate of Correctness (PCC) method, as presented in [61], is used to analyze the design configurations to decide which configurations have the highest probability to meet the requirements. The PCC driver model utilizes uncertainty propagation methods to compute the PCC value for each output (i.e.,

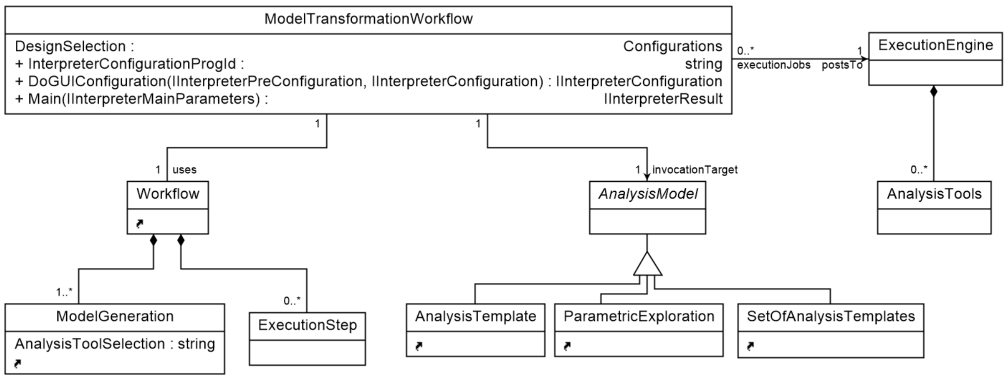

ModelTransformationWorkflow
DesignSelection : Configurations
+ InterpreterConfigurationProgId : string
+ DoGUIConfiguration(IInterpreterPreConfiguration, IInterpreterConfiguration) : IInterpreterConfiguration
+ Main(IInterpreterMainParameters) : IInterpreterResult
0..*
executionJobs
postsTo
1
ExecutionEngine
1
1
1
uses
Workflow
1
invocationTarget
AnalysisModel
0..*
AnalysisTools
1..*
ModelGeneration
AnalysisToolSelection : string
0..*
ExecutionStep
AnalysisTemplate
ParametricExploration
SetOfAnalysisTemplates

Figure 16: Tool integration class diagram

metric). The PCC driver model defines the probability density function for the inputs (i.e., parameters) of the analysis template. The supported density functions are: (1) beta distribution, (2) log normal distribution, (3) normal distribution, and (4) uniform distribution. These input values are sampled according to the driver method and the probability density function. The supported uncertainty propagation methods are: (1) Monte Carlo Simulation (MCS), (2) Taylor Series Approximation (TS), (3) Most Probable Point Method (MPP), (4) Univariate Dimension Reduction Method (UDR), and (5) Polynomial Chaos Expansion (PCE). A detailed description of each method is presented in [61]. The output values of the system must be in between the minimum and maximum values that are defined by the PCC driver model. For each output a target PCC value is defined, which specifies the acceptable probability for the output value to fall in between the minimum and maximum value. In order to compare configurations effectively, a joint PCC value is computed, which is a real number between 0 and 1. The results of the uncertainty propagation are probability density functions for each output (i.e., metric). In addition to the uncertainty propagation, sensitivity analysis is also supported by the PCC driver model.

Tool integration and analysis package execution

In this section we present our solution for tool integration and analysis package execution. The Tool Integration Platform bridges the gap between the Model Integration Platform and Execution Integration Platform. The Tool Integration Platform has two main aspects: (1) extracting models from external model based system engineering tools and (2) generating analysis packages from executable analysis models through model transformations. Each aspect and domain of the design model is associated with one or more model transformations. These model transformations are implemented to work with individual configurations and generate executable models from analysis templates. In order to reuse these model

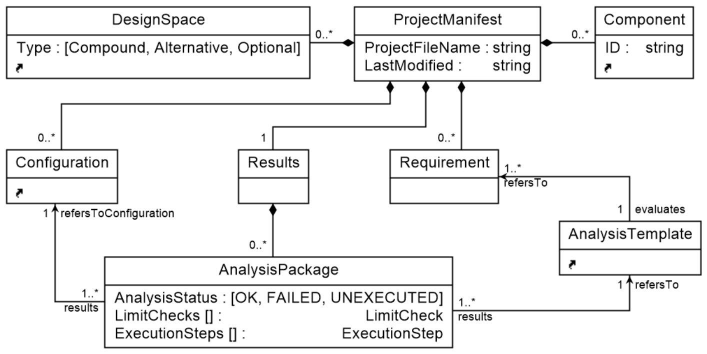


Figure 17: Project data structure class diagram

transformations, a *Master Interpreter* is created that handles analysis templates specified for design spaces and creates analysis packages from the extracted domain models by using the domain model transformations. Figure 16 depicts the UML class diagram for this tool integration. The model transformation workflow is implemented by the *Master Interpreter* in the OpenMETA tools. This model transformation workflow requires a few key concepts to be defined: (a) the workflow, (b) the analysis model, and (c) the execution engine.

A workflow defines a set of model generation steps and execution steps to generate the analysis packages and execute the analyses (e.g., simulation or verification), respectively. It refers to an analysis model, which can be: an analysis template, a parametric exploration, or a set of analysis templates for a configuration or a design space. This means that each analysis model can be defined for a design space and the model transformation workflow (i.e., *Master Interpreter*) will transform all configurations for individual analysis packages. When analysis packages are generated, the model transformation workflow creates an execution job and posts it to the *execution engine*. Execution engine is a software that executes the analysis packages. The execution engine must have all the required analysis tools installed and deployed. The execution engine is a part of the Execution Integration Platform and in the OpenMETA tools it is called the *Job Manager*. The Job Manager supports local and cloud-based analysis package execution to utilize all available resources.

Project artifacts and analysis result management

Product design involves the management of design artifacts and analysis results. In Model-Based Systems Engineering (MBSE), the primary artifacts are models, which are stored in a model database. Figure 17 shows a UML class diagram that represents how our project artifacts are organized. The project manifest is the entry point of the project data that stores project-level properties such as the file name of the project or the last modified date. The project manifest captures: design space models, configurations, a single *results* object, requirements, and component models. For each configuration and analysis template pair, an analysis package is generated that maintains the dependency to the configuration and to the selected analysis tool. The analysis package indicates the analysis status: unexecuted, failed, or succeeded. In addition to this high-level information, it contains a list of execution steps and a list of limit violations. Limit checking objects are defined in the domain models to identify limit violations that often represent physical limits of the component (e.g., the maximum torque on a physical part cannot exceed a certain value.)

Because each analysis package refers to an analysis template, they indirectly refer to one or more requirements. The Visualization Integration Platform uses this project data structure to discover and visualize the analysis results, configurations, and requirements about the designed system. The visualization of the results is implemented by the *Project Analyzer* in the OpenMETA tools.

Evaluation

In this chapter we contributed to all four platforms that we presented in Chapter III. Our solution supports executable requirements for designs and design spaces in the form of analysis template models. The analysis template models can be reused for parametric design space exploration and for grouping analyses into a set of analysis templates. Analysis templates, parametric exploration models, and sets of analysis templates are translated to executable analysis packages. Analysis packages are executed by an execution engine provided by the Execution Integration Platform. The generated analysis packages and analysis results are organized and structured for the Visualization Integration Platform.

CHAPTER VI

Analysis-driven Rapid Design Process

Problem Statement

In addition to the reusable analysis templates and execution framework, the requirement trade-offs must be evaluated rapidly for several possible design candidates in an iterative design process. The Analysis-driven Rapid Design Process (ARDP) combines the analysis templates with discrete and parametric design space exploration. The discrete design space exploration provides architecture and component choices with static constraints; whereas the parametric design space exploration allows designers to sweep component, design, or environment parameters to generate a *surrogate model*, perform sensitivity analyses, or optimize the final design candidates. A surrogate model is a simplified model that substitutes the system model with a representation of the system behavior in a valid parametric range. A surrogate model is often used when a detailed model exists but its execution takes excessive time. Existing analysis templates should be reusable without extra effort from the designers in both discrete and parametric design spaces. As a result of implementing this ARDP the design time will be significantly reduced as described in the evaluation section below.

Challenges

Designing products often involves a high dimensional design space, which creates several challenges: (a) how to eliminate the unfeasible portion of the design space effectively, (b) how to capture constraints and apply them incrementally on the design space, and (c) how to determine which analysis will yield the most significant design space reduction. Hard and soft requirements are evaluated by various analysis types: how to group these analyses? The analyses for CPS design are similar concepts to unit and functional tests in software development (e.g., test-driven development). In software development the tests are often run automatically by a continuous integration system to easily identify problems in all design phases. This leads to a research challenge: how to reduce development time for CPS design, and effectively use computational resources without eliminating possible design solutions?

Solution

In this section we present a design flow for an Analysis-driven Rapid Design Process (ARDP) that utilizes all four platforms. Important to note that there are other possible

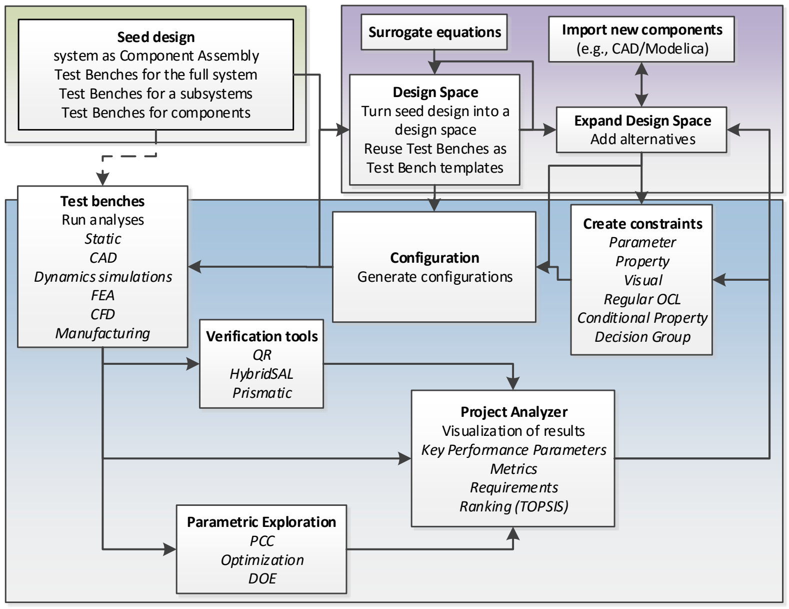


Figure 18: Analysis-driven Rapid Design Process: design flow

design flows can be implemented. We assume that a library of component models, an initial design, and simulations or analyses are given in a Model-Based Systems Engineering (MBSE) tool. The component models are imported into the OpenMETA tools by using the domain model importers. The initial design is used as a seed design and created in the OpenMETA tools as a *CyPhy Component Assembly*. The simulation and analyses models are converted to analysis templates called *CyPhy Test Benches*.

Figure 18 shows that our initial setup including all analyses can be executed on the seed design. At this point of the design process it is irrelevant if the initial design meets the system requirements or not. However, it is important to get a working model that can be used to seed our discrete design space. There is automated tool support to turn the seed design into a design space that yields a single configuration.

OpenMETA tools can be used to create new component instances if the imported component models are parametric. These new instances can be added as component choices to the design space to explore more configurations. In addition to these component choices, alternative topologies can be represented by this hierarchical design space model. By implementing and applying design constraints, unfeasible configurations are effectively eliminated that would violate those constraints. These features provide management and continuous refinement of the design space.

The existing *CyPhy Test Benches* can be configured to work on the entire design space as long as the design space provides the same interfaces that the *CyPhy Component Assembly* (seed design) provides. If the design space has the same interfaces, then each generated configuration will have the same interfaces. This approach minimizes the effort to evaluate the analysis templates for an entire design space by reusing the original model.

Some analysis templates may take more time to evaluate than others. It is a good practice to group the analysis templates by execution time and requirement groups. For instance, if our potential design space is large (e.g., contains several hundreds of configurations), it is not preferable to start with FEA simulations that could easily take several hours per configuration. It is better to evaluate cost, weight, or spatial dimensions (height, length, or width); or to group the analysis templates based on hard and soft requirements. These analyses often takes only a few seconds to a minute per configuration.

Our design environment promotes model reuse, therefore the analysis templates can be used by formal and probabilistic verification tools as well as parametric design space exploration models. All analysis results are collected and consolidated for the *Project Analyzer* as shown in Figure 18. The *Project Analyzer* provides visualization and ranking techniques to explore the generated results for both discrete and parametric design spaces. Based on our design decisions, one could: (1) add or relax design space constraints and (2) add or remove component choices and topologies from the design space to find the best configuration for our designed product. One of the key elements in our ARDP is this built-in feedback loop in the design flow, which makes rapid design iteration possible.

The analysis templates can also be used for unit (component) testing and integration (design model) testing. The automated execution of analysis templates can be triggered by changes made to CPS components and designs, which is a similar concept to continuous integration systems in software development. These presented features make an iterative and rapid design process possible.

Evaluation

We developed two use cases to evaluate and utilize our Analysis-driven Rapid Design Process (ARDP): (a) an oscillator circuit design and (b) a ground vehicle driveline design. The oscillator design represents a simplified system to show all design phases in the design process. The ground vehicle driveline design model represents a Cyber-Physical System (CPS) of more realistic size and scale. In the simplified case, we compare the time taken to build and analyze the models for the system using a Modelica simulation tool and the OpenMETA tools. The oscillator design uses only the electrical domain and a Modelica simulation tool.

In addition to these two examples, an earlier version of the META engineering design process was evaluated in [40] by using a sophisticated System Dynamics (SD) model, which

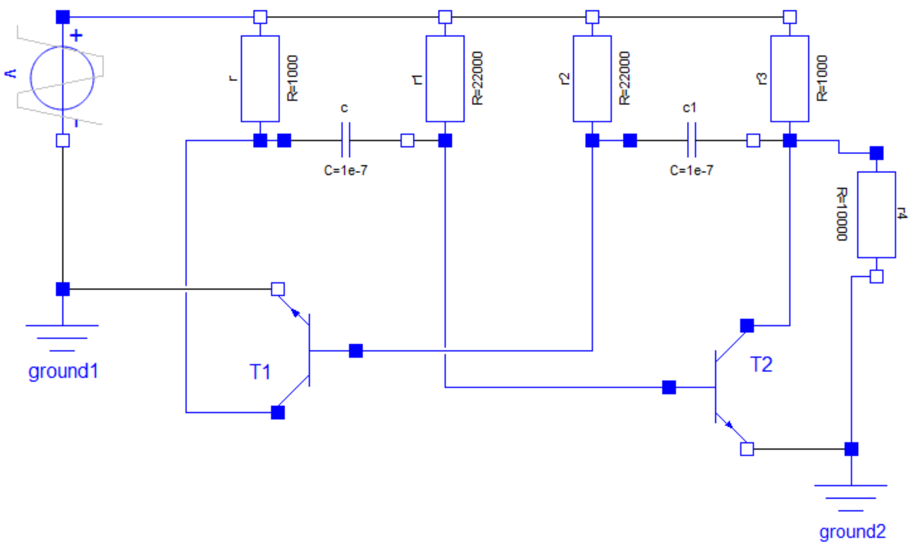


Figure 19: Oscillator example: `Modelica.Electrical.Spice3.Examples.Oscillator` [6]

allows the simulation of the design process. The study demonstrated a speedup factor of 4.4 for this process compared to current practice against a benchmark case with 3,000 requirements. The results were also validated against data from the Boeing 777 Electric Power System (EPS) design project where a speedup factor of 3.8 was achieved.

Oscillator

In this section, we present a simple electrical circuit design problem to demonstrate and evaluate our Analysis-driven Rapid Design Process (ARDP). The purpose of this use case is not to compare the design process with existing circuit design tools, but rather to demonstrate each phase of this design process on a simple use case. For this use case we selected an oscillator circuit (i.e., astable multivibrator) from the Modelica Standard Library (MSL) as-is `Modelica.Electrical.Spice3.Examples.Oscillator`. Furthermore, we highlight under what assumptions and conditions this design process is more effective than the traditional design process. In addition to the aforementioned goals, we aim to identify issues as early in the design process as possible.

$$v(t) = \begin{cases} 0\text{V}, & 0 < t < 0.5\text{ms} \\ \frac{8\text{V}}{10\text{ms}}t - 0.4\text{V}, & 0.5\text{ms} < t < 10.5\text{ms} \\ 8\text{V}, & 10.5\text{ms} < t \end{cases} \tag{10}$$

Figure 19 depicts the selected design model and its testing environment setup in Modelica. The oscillator design contains two transistors denoted by T_1 and T_2, two capacitors denoted

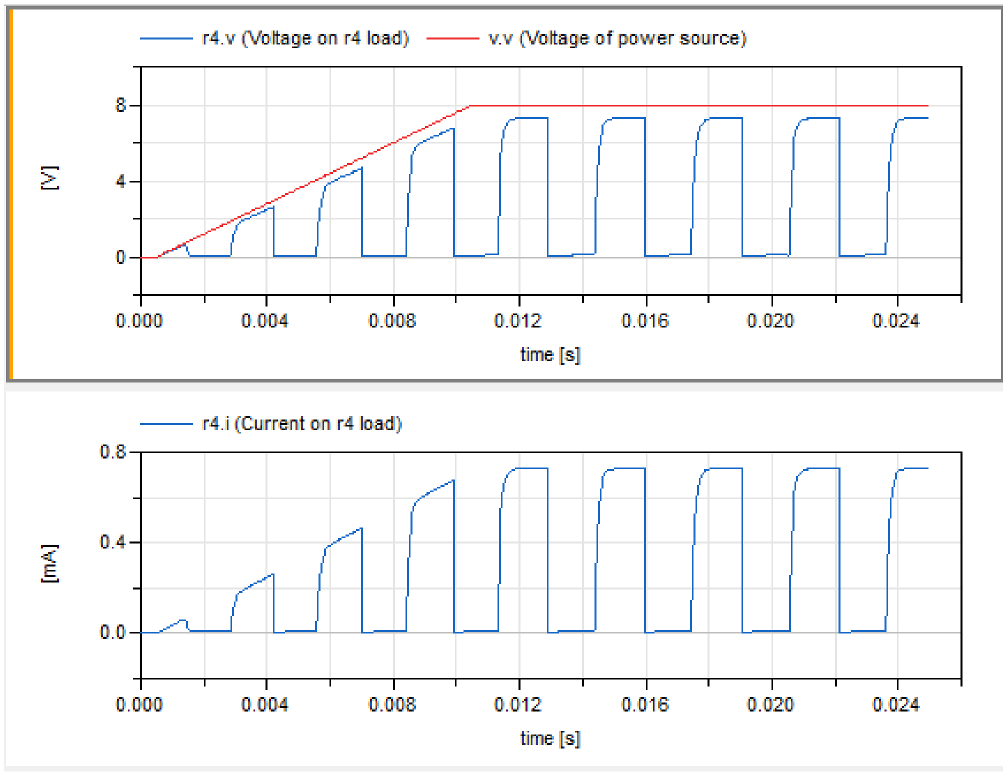


Figure 20: Oscillator example: simulation results

by $c = 100\text{nF}$ and $c_1 = 100\text{nF}$, and four resistors: a base resistor for each transistor $r_1 = 22\text{k}\Omega$ and $r_2 = 22\text{k}\Omega$ and a collector resistor for each transistor $r = 1\text{k}\Omega$ and $r_3 = 1\text{k}\Omega$. There are four elements in the circuit diagram that are associated with the test definition: (a) the voltage source denoted by v, (b) the two grounds denoted by *ground1* and *ground2*, and (c) the resistive load denoted by $r_4 = 10\text{k}\Omega$. The voltage source drives T_1 transistor in the oscillator with a ramp function as shown in Eq. 10. After the voltage source reaches its maximum value 8V the output signal shape becomes periodic. The two grounds constrain the voltage to 0V on the emitter for both transistors. The collector and emitter of T_2 transistor is connected to the resistive load.

A Modelica simulation tool called Dymola [144] was used to execute the simulation for this model. The original simulation setup was as follows: (a) the selected solver is the differential/algebraic system solver (DASSL) [125], (b) the start time is 0s, (c) the stop time is 25ms, (d) the tolerance is 10^{-4}, and (e) the number of intervals is 500. The simulation results are shown in Figure 20, where the plot on the top depicts the voltage across the resistive load $r_4.v(t)$ and the ramp voltage of the power source $v.v(t)$. The plot on the bottom shows the current through the resistive load $r_4.i(t)$.

$$\begin{aligned} v_{4_{max}} &= \frac{r_4}{r_4 + r_3} v_{max} \\ v_{4_{max}} &= \frac{10\text{k}\Omega}{10\text{k}\Omega + 1\text{k}\Omega} 8\text{V} \\ v_{4_{max}} &= \frac{10\text{k}\Omega}{11\text{k}\Omega} 8\text{V} \\ v_{4_{max}} &\simeq 7.27\text{V} \end{aligned} \tag{11}$$

When the T_2 transistor is open (i.e., OFF), there is a voltage divider for the output voltage that determines the maximum voltage across the load $v_{4_{max}}$ and maximum current through the resistive load. According to Eq. 11 the maximum voltage $V_{max} = v_{4_{max}}$ is 7.27V and as Eq. 12 shows the maximum current $I_{max} = i_{4_{max}}$ is 727µA.

$$\begin{aligned} i_{4_{max}} &= \frac{v_{4_{max}}}{r_4} \\ i_{4_{max}} &\simeq \frac{7.27\text{V}}{10\text{k}\Omega} \\ i_{4_{max}} &\simeq 727\mu\text{A} \end{aligned} \tag{12}$$

This initial design model is used as our seed design. In the following sections we define a set of requirements, explore alternative options for the transistor components T_1 and T_2, explore resistor alternatives for r and r_2, and explore capacitor alternatives c and c_1 to find the most feasible design configurations that meet all requirements.

After selecting the feasible design candidates, we will test the robustness of the design using parametric design space exploration utilizing the Probabilistic Certificate of Correctness (PCC) and Design of Experiment (DoE) capabilities of this Analysis-driven Rapid Design Process. Finally, we present an evaluation on how much time it took to build and analyze this oscillator system, compare it with using only the model-based engineering tool (i.e., Modelica), and conclude under what conditions this design process is more effective than the traditional design process.

Requirements

Table 2 summarizes all of our design requirements for this oscillator design problem. Each requirement defines a threshold and an objective value; the designed system must pass the threshold to meet the requirement. There are three main design categories: (1) timing properties, (2) currents, and (3) voltages. The timing properties category contains three requirements: (1) rise time t_r, (2) fall time t_f, and (3) frequency f. The currents

Category	Name	Symbol	Threshold	Objective
Timing properties	Rise time	t_r	280µs	240µs
Timing properties	Fall time	t_f	15µs	13µs
Timing properties	Frequency	f	325Hz	450Hz
Currents	Maximum Current	I_{max}	800µA	750µA
Currents	Minimum Current	I_{min}	4µA	3µA
Voltages	Maximum Voltage	V_{max}	7.7V	7.4V
Voltages	Minimum Voltage	V_{min}	40mV	30mV

Table 2: Oscillator: Requirements

category has two requirements: (1) maximum current I_{max} and (2) minimum current I_{min}. The voltages category has two requirements: (1) maximum voltage V_{max} and (2) minimum voltage V_{min}. All requirements are evaluated based on the example model and the numerical values are measured on the resistive load r_4.

Alternative components

For this design problem, we consider component alternatives for the base resistors r and r_2, the capacitors c and c_1, and the transistors T_1 and T_2, The default resistor value is 22kΩ for r and r_2 called R_B; and the five resistor alternatives are shown in Table 3. The default capacitor value is 0.12µF for c and c_1 called C; and the five capacitor alternatives are shown in Table 4.

Base Resistors (R_B)
18kΩ
20kΩ
22kΩ
24kΩ
27kΩ

Table 3: Oscillator: Resistor alternatives

Capacitor (C)
0.068µF
0.082µF
0.12µF
0.15µF
0.1µF

Table 4: Oscillator: Capacitor alternatives

Table 5 lists all parameters that are considered for each transistor. The transistor alternatives and their parameter values are summarized in Table 6. Parameter values are rounded to fit here by a maximum of 0.3% error and only the key parameters are shown because of space limitations. The selected transistors cover a broader application range than what is required for oscillator design, but the purpose of this use case is to demonstrate the design process.

Symbol	**Unit**	**Definition**
I_S	A	Transport saturation current
B_F		Ideal maximum forward beta F
B_R		Ideal maximum reverse beta
V_A	V	Early voltage
R_B	Ω	Zero bias base resistance
R_C	Ω	Collector resistance
R_E	Ω	Emitter resistance
T_F	s	Ideal forward transit time
T_R	s	Ideal reverse transit time
C_{JE}	F	Zero bias B-E depletion capacitance
M_{JE}		B-E junction exponential factor
V_{JE}	V	B-E built in potential
C_{JC}	F	Zero bias B-C depletion capacitance
M_{JC}		B-C junction grading coefficient
V_{JC}	V	B-C built in potential

Table 5: Oscillator: Transistor parameters

Type	I_S	B_F	B_R	V_A	R_B	R_C	R_E	T_F	T_R
2N2222	248fA	100	5	73.9V	0.13Ω	0.3Ω	0.4Ω	400ps	40ns
2N3055	4.66pA	60	2	100V	3Ω	0.44Ω	1mΩ	99.5ns	570ns
2N3904	6.734fA	140	4	74V	10Ω	1Ω	0Ω	301ps	240ns
2N4401	9.09fA	300	4	113V	1.27Ω	0.127Ω	0.32Ω	512ps	151ns
2N5179	0.069fA	282.1	1.176	100V	10Ω	4Ω	0Ω	141ps	1.59ns
2N5551	2.511fA	155	3.197	100V	10Ω	0Ω	0Ω	560ps	1.2ns
2SC4793	1.8pA	146.38	4	273V	1.7Ω	0Ω	0Ω	1.23ns	983ns
2SC5200	30.46pA	96.2	4.849	100V	20.18Ω	0Ω	0Ω	686ps	10ns
BC547A	15.3fA	180	8.628	69.7V	1Ω	0.65Ω	0.64Ω	500ps	1fs
BC550C	45fA	200	12.2	162V	167Ω	0Ω	0Ω	595ps	10ns
BC847A	10.2fA	180	4	121V	3.66Ω	0Ω	0Ω	427ps	50.3ns
MJE340	801fA	123.09	0.00419	100V	4.44mΩ	0Ω	0Ω	16.5ns	10ns

Table 6: Oscillator: Transistor alternatives [33]

By using the information about these alternative components we developed a component library to capture all component alternatives. To verify that the simulation results are sensitive to component choices a few alternative designs were manually created and simulated using a Modelica simulation tool. As a result of having five resistor alternatives, five capacitor alternatives, and twelve transistor alternatives our combinatorial design space yields $90,000$ possible configurations according to Eq. 13.

$$\begin{aligned} \mathit{numberOfConfigurations} &= 12(T_1) \times 5(r) \times 5(c) \times 12(T_2) \times 5(r_2) \times 5(c_1) \\ \mathit{numberOfConfigurations} &= 90,000 \end{aligned} \tag{13}$$

Clearly, manually exploring this design space is labor intensive, error-prone, time consuming, and wasteful in terms of computational resources. The $90,000$ configurations contain several configuration clusters that are outside of our interest. For instance, when different transistor instances are used for T_1 and T_2, or when the capacitor value is different for c and c_1. However, Modelica supports *replaceable* model elements that can be used to overcome this issue. Replaceable elements constrain the component choices and keep the two transistors the same. Note that not all model-based engineering tools have such a feature. Now consider a more complicated constraint, which is based upon parameter values from the selected components; such a case cannot be expressed using the Modelica language. The purpose of this use case is not to show why our design approach and tools are better than Modelica, rather *when* to use Modelica only and *when* to use our design approach.

Discrete design space exploration

Instead of exploring the discrete design space in Modelica, the entire component library was imported into the OpenMETA tools [69] and the design was developed according to the original Modelica model. Using this design model an analysis template was designed to evaluate all seven requirements of the designed system. Figure 21 depicts an executable requirement model in the form of an analysis template. This analysis template contains a reference to the system design, and defines the environment including test drivers and loads, input parameters, solver settings, post processing of the raw result data, and output metrics that are directly associated with the requirements.

It is often important to keep analyses of the results outside of the simulation model, and process the generated time series data afterward instead of putting more calculations on the dynamics solver. If the data processing for results analysis involves higher-order equations or defines new variable derivatives, it could negatively impact the simulation execution time due to the increased number of equations. Therefore, we do *post processing* of results outside the

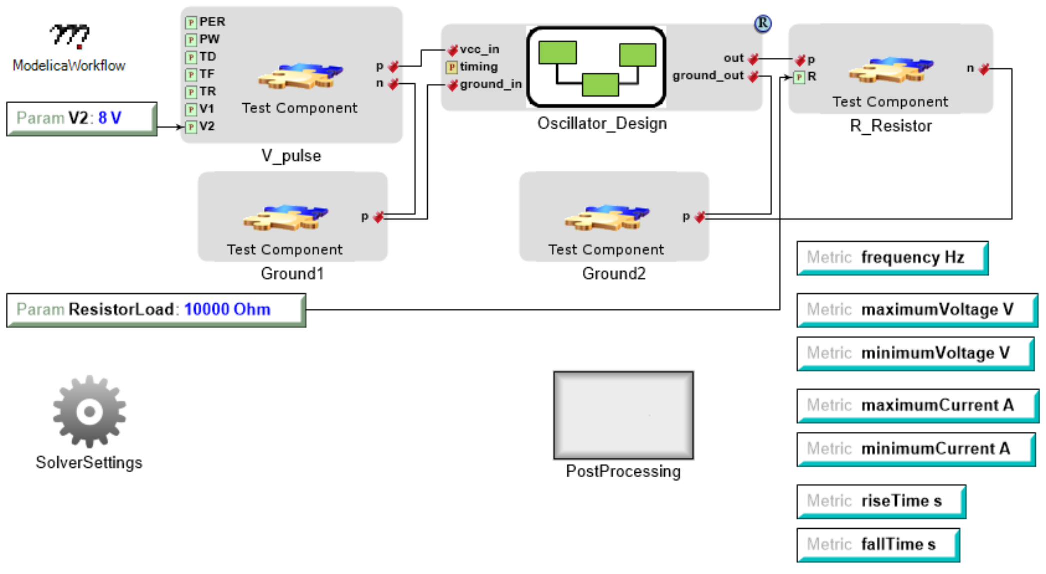


Figure 21: Oscillator: Analysis template

simulation. For instance, if each simulation takes one extra second, simulating a design space with $90,000$ configurations would add ~ 25h to the total execution time. If the execution is sequential due to limited resources, the design process would be delayed by an entire day.

When the analysis template model becomes functional and executable, our original design (i.e., seed design) can be turned into a design space. The resulting design space is shown in Figure 22. As we discussed before, there are alternative component choices for six components in the design. The Design Space Exploration and Refinement Tool (DESERT) [107] allows us to prune the design space using several design constraints. In this example, a total of five constraints are used: three visual constraints and two property constraints. These visual constraints express the symmetric relationship between the transistors (T_1, T_2), the resistors (r, r_2), and capacitors (c, c_1), which means that pairwise they must be the same instance (i.e., have the same parameters). The two property constraints represent a limit on the minimum and maximum value (Eq.14) of the R_BC circuit, which is directly related to the timing property of the oscillator.

$$\begin{aligned} 2.2\text{ms} \leq R_BC \\ R_BC \leq 3\text{ms} \end{aligned} \tag{14}$$

By applying all five constraints, the design space yields 84 viable configurations. Assuming that the execution of the analysis template takes one second per configuration, these 84 configurations would take 84s (~ 1.5min) vs ~ 25h for all $90,000$. First, we wanted to verify that all transistor options can be simulated without wasting computational time; the model

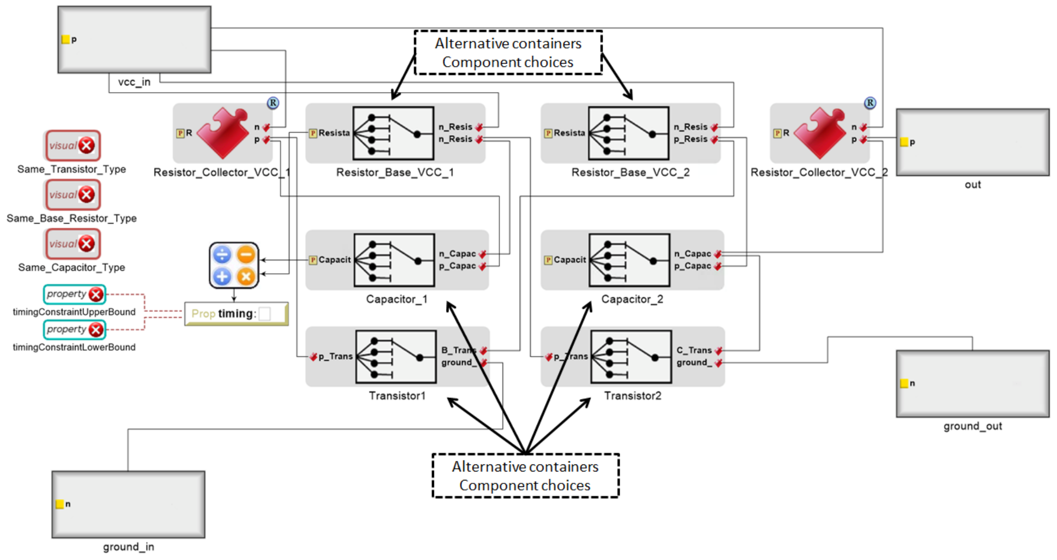


Figure 22: Oscillator: Design space

for the transistor is more complex than for the resistor and the capacitor, which could cause solver errors. Therefore, we fixed the resistor and the capacitor choices to a single instance, and considered only a subset of the design space with 12 transistor alternatives. Out of the 12 simulations one configuration failed with a simulation error as shown in Appendix C.

$$\begin{aligned} \mathit{Start\ time} &= 0\text{s} \xrightarrow{\mathit{unchanged}} 0\text{s} \\ \mathit{Stop\ time} &= 25\text{ms} \xrightarrow{\mathit{changed\,to}} 60\text{ms} \\ \mathit{Number\ of\ intervals} &= 500 \xrightarrow{\mathit{changed\,to}} 5000 \\ \mathit{Tolerance} &= 10^{-4} \xrightarrow{\mathit{changed\,to}} 10^{-6} \end{aligned} \tag{15}$$

After changing the solver settings as described by Eq. 15 all 12 configurations are successfully simulated. Next, we executed all 84 configurations for the viable design space; 4 configurations failed (cfg51, cfg53, cfg55, and cfg81) because of the solver settings. Selecting and fine-tuning the dynamics simulation solver is outside of the scope of this thesis, but we have learned that this Design Space Exploration and Refinement Tool can be used with analysis templates to quickly discover such issues (e.g., modeling errors or simulation setup errors) in the early phases of the design process.

Ranking designs by simulation results

In order to effectively compare the configurations, weights are assigned for each requirement, which are used in the scoring function by the TOPSIS analysis. The TOPSIS analysis

Symbol	Metric name	Min/Max	Weight $(0-1)$
f	frequency	max	0.68
t_f	fall time	min	0.50
t_r	rise time	min	1.00
I_{min}	minimum current	min	0.65
I_{max}	maximum current	min	0.50
V_{min}	minimum voltage	min	0.77
V_{max}	maximum voltage	min	0.50

Table 7: Oscillator: Scoring weights

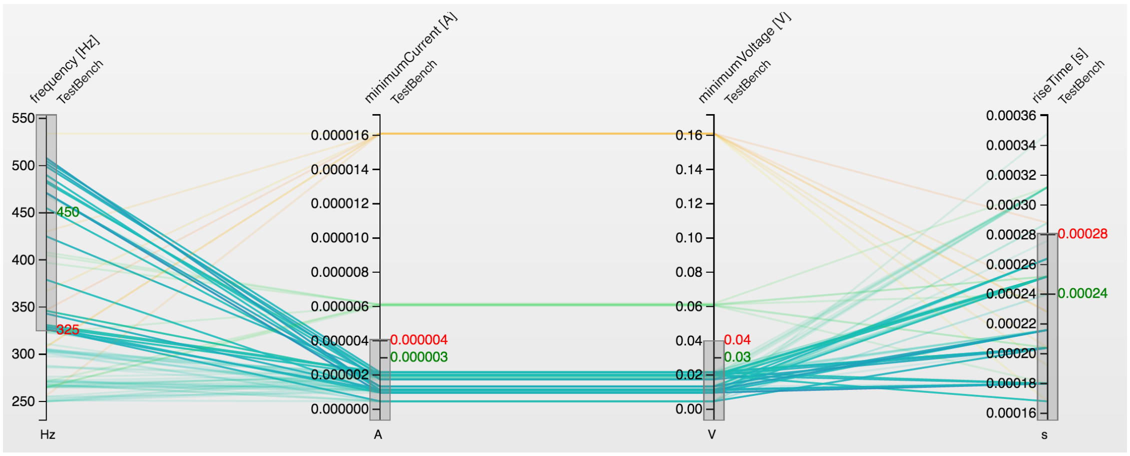


Figure 23: Oscillator: Parallel axis plot colored by ranking

assigns a value between 0 and 1 for each configuration based on the weights shown in Table 7. The configurations are ranked and sorted in descending order based on the scores. The detailed results including component choices, requirements (Passed Objective, Passed Threshold, or Failed), numerical values, and the score are shown in Appendix D. The maximum voltage $V_{max} \simeq 7.27$V and maximum current $I_{max} \simeq 727$µA are omitted from the results because all configurations have the same values. The minimum voltage V_{min} is also omitted from the results because it can be easily derived from the minimum current I_{min}. For instance, for cfg39 $I_{min} \simeq 0.969$µA, which results in $V_{min} = I_{min} r_4 \simeq 0.969\text{µA} \times 10\text{k}\Omega \simeq 9.69$mV.

The numerical results are presented in Appendix D for all configurations. The Project Analyzer provides a visualization technique to show each configuration and the associated numerical values in a single plot called parallel axis plot. The parallel axis plot depicts four selected axes: (a) frequency, (b) minimum current, (c) minimum voltage, and (d) rise time for all configurations in Figure 23. The requirement threshold and objective values are shown on each vertical axis. Each line in the plot is associated with a single configuration in the

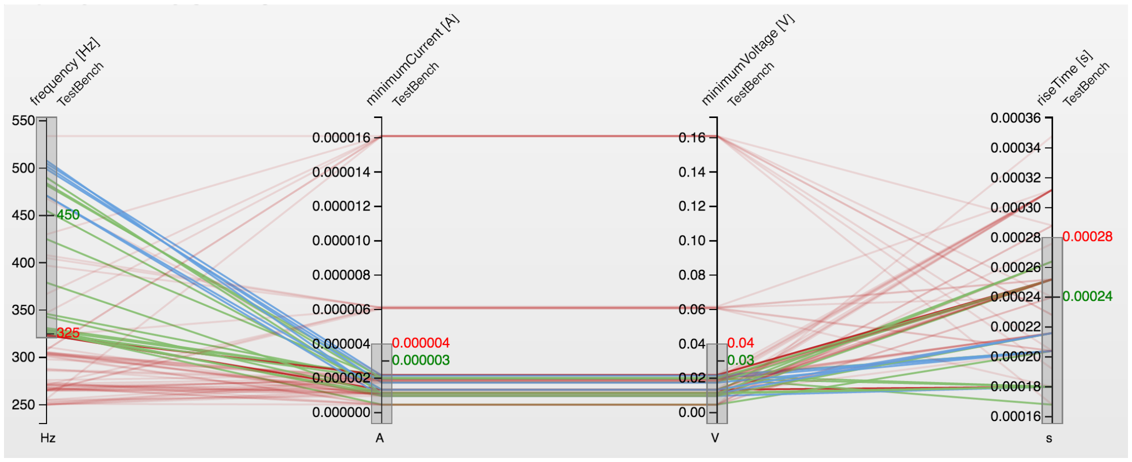


Figure 24: Oscillator: Parallel axis plot colored by requirements

design space. The color of the lines varies according to the score, where red is 0 and blue is 1. Figure 24 depicts another parallel axis plot, where each configuration is color coded based on the requirements: (a) red means the configuration failed one or more requirements, (b) green means the configuration passed all threshold values for all requirements, (c) blue means the configuration passed all objective values for all requirements, and (d) gray means some of the requirement values are not available for the configuration.

Parametric design space analysis

The numerical results presented in the previous section correspond to the nominal parameters for the components and the environment (e.g., resistive load). Even though some configurations had higher scores than others, those configurations may not be as robust or stable to meet the requirements if there is uncertainty on the parameter values. In order to analyze the output changes for each configuration, two parametric design space analyses are performed: (1) a Probabilistic Certificate of Correctness (PCC) analysis and (2) a Design of Experiment (DoE) analysis. The PCC method, as presented in [61], is used to analyze the design configurations to decide which configurations have the highest probability to meet the requirements. The PCC value is a number between 0 and 1, which represents the probability that the outputs will fall within our specified ranges.

The existing analysis template, called a Test Bench, is reused as-is in a PCC model as shown in Figure 25. The PCC model contains two models elements: a Test Bench model and a PCC driver model. The PCC driver records four out of seven outputs of the Test Bench model and defines confidence intervals for them: (1) frequency (325Hz – 450Hz, target value: 85%), (2) minimum current (0A – 4µA, target value: 80%), (3) minimum voltage (0V – 40mV, target value: 75%), and (4) rise time (0s – 280µs, target value: 92.5%). The PCC driver

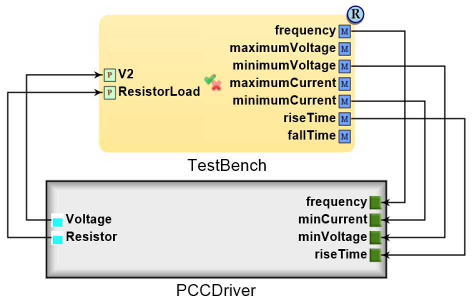
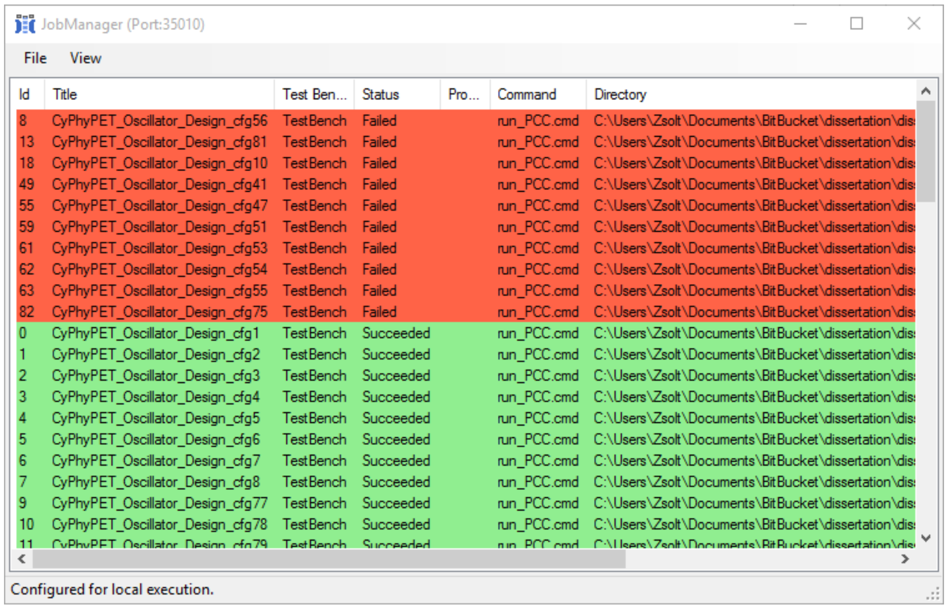


Figure 25: Oscillator: Probabilistic Certificate of Correctness model



Id	Title	Test Ben...	Status	Pro...	Command	Directory
8	CyPhyPET_Oscillator_Design_cfg56	TestBench	Failed		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
13	CyPhyPET_Oscillator_Design_cfg81	TestBench	Failed		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
18	CyPhyPET_Oscillator_Design_cfg10	TestBench	Failed		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
49	CyPhyPET_Oscillator_Design_cfg41	TestBench	Failed		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
55	CyPhyPET_Oscillator_Design_cfg47	TestBench	Failed		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
59	CyPhyPET_Oscillator_Design_cfg51	TestBench	Failed		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
61	CyPhyPET_Oscillator_Design_cfg53	TestBench	Failed		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
62	CyPhyPET_Oscillator_Design_cfg54	TestBench	Failed		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
63	CyPhyPET_Oscillator_Design_cfg55	TestBench	Failed		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
82	CyPhyPET_Oscillator_Design_cfg75	TestBench	Failed		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
0	CyPhyPET_Oscillator_Design_cfg1	TestBench	Succeeded		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
1	CyPhyPET_Oscillator_Design_cfg2	TestBench	Succeeded		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
2	CyPhyPET_Oscillator_Design_cfg3	TestBench	Succeeded		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
3	CyPhyPET_Oscillator_Design_cfg4	TestBench	Succeeded		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
4	CyPhyPET_Oscillator_Design_cfg5	TestBench	Succeeded		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
5	CyPhyPET_Oscillator_Design_cfg6	TestBench	Succeeded		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
6	CyPhyPET_Oscillator_Design_cfg7	TestBench	Succeeded		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
7	CyPhyPET_Oscillator_Design_cfg8	TestBench	Succeeded		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
9	CyPhyPET_Oscillator_Design_cfg77	TestBench	Succeeded		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
10	CyPhyPET_Oscillator_Design_cfg78	TestBench	Succeeded		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis
11	CyPhyPET_Oscillator_Design_cfg79	TestBench	Succeeded		run_PCC.cmd	C:\Users\Zsolt\Documents\BitBucket\dissertation\dis

Configured for local execution.

Figure 26: Oscillator: Failed Probabilistic Certificate of Correctness configurations

varies two inputs with a probabilistic distribution function: (1) the maximum voltage of the voltage source with a uniform distribution (6V – 10V) and (2) the value of resistive load with a uniform distribution (9.5kΩ – 10.5kΩ, considering 5% resistor precision). The PCC driver uses univariate dimension reduction method [130] for the uncertainty propagation in the model and generates input values based on the specified distributions.

All 84 configurations are executed for the PCC model. Figure 26 depicts 10 configurations that failed to execute successfully. The cause of the failure originated from the selected dynamics system solver. As we mentioned before, selecting and fine-tuning the dynamics solver is outside of the scope of this thesis. By analyzing the simulation results, cfg39 and cfg72 have the highest score in ranking, but cfg3 and cfg45 have the highest PCC value. Figure 27 depicts the output distributions, PCC value, and complexity value for all four configurations and four variables. Each column is associated with one configuration in Figure 27 in the order as follows: cfg39, cfg72, cfg3, and cfg45. Each row represents

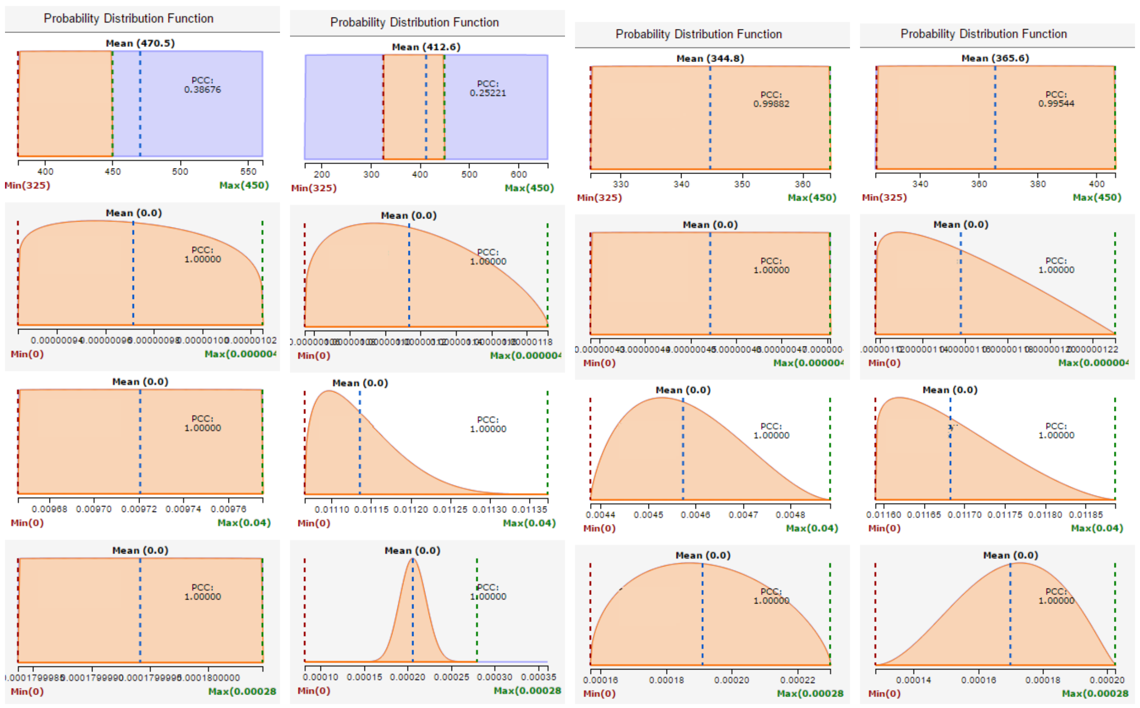


Figure 27: Oscillator: PCC results for cfg39, cfg72, cfg3, and cfg45

an output of the Test Bench model in the order as follows: frequency, minimum current, minimum voltage, and rise time. Even though cfg39 and cfg72 are the best configuration choices according to the ranking, they are not as robust as cfg3 and cfg45, which still meet all design requirements. Figure 28 shows the overlaid time series simulation results of the selected four configurations.

The PCC model can be easily extended to consider parametric variations and uncertainty for each resistor and capacitor component in the design. By executing that modified model we can determine if we can use resistors with a 5% tolerance, or if it is necessary to use 1% components.

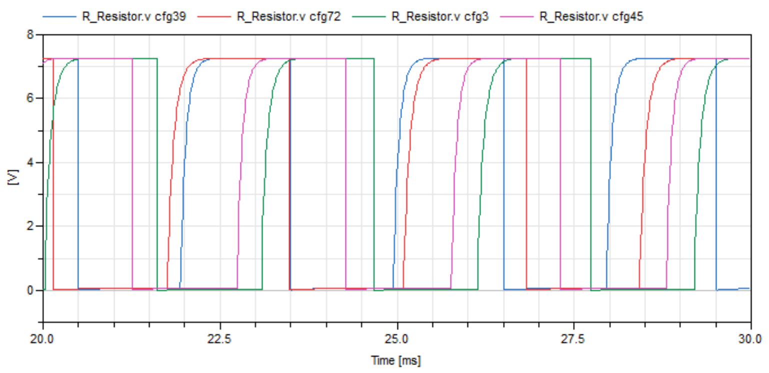


Figure 28: Oscillator: Overlaid simulation results

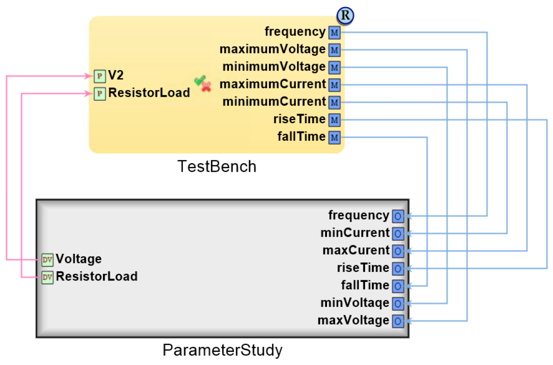


Figure 29: Oscillator: Design of Experiment model

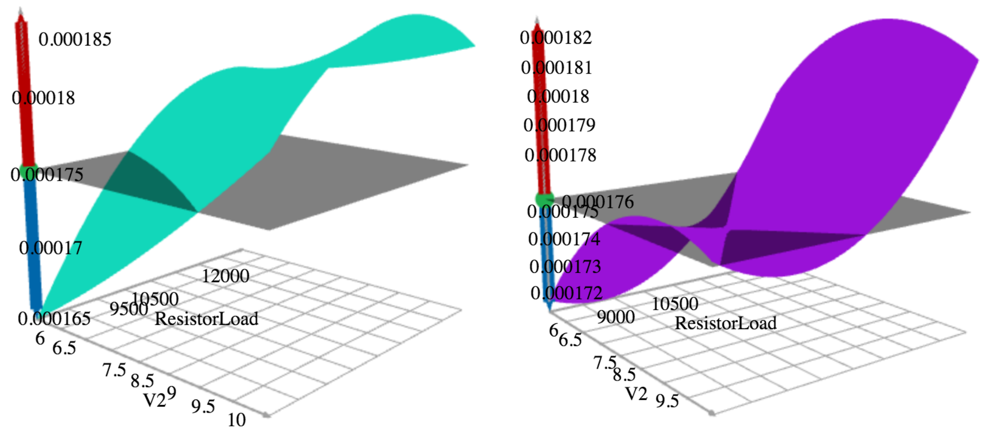


Figure 30: Oscillator: Design of Experiment results for rise time (cfg39 and cfg45)

The highest ranking design is cfg39 and the most robust design is cfg45. On these two configurations another parametric design space analysis is performed. The exact same analysis template (i.e., Test Bench) is used to set up a Design of Experiment (DoE) model to create a parameter study. Figure 29 depicts the DoE model, where the value of the resistive load and the maximum voltage of the voltage source are sampled within the parametric space by the selected DoE method called full factorial. The full factorial driver uses 5 levels, i.e., 5 samples per input, which is a total of 25 samples per configuration. In addition to the DoE driver a surrogate model is generated using the response surface methodology. Figure 30 shows the resulting response surfaces for cfg39 and cfg45. The 3D plots correspond to the variations made to the resistive load and to the maximum voltage on the voltage source, and shows the results for rise time on the vertical axis.

Evaluation

In order to evaluate our design process, the time to perform the key activities involved are measured and presented in Table 8. By using this Analysis-driven Rapid Design Process (ARDP) along with the design flow, as presented in this chapter, it took approximately 10 hours and 22 minutes to design this oscillator circuit. The design activities include: (a) creating the requirements, (b) building a component library, (c) exploring possible design alternatives and component choices, and (d) analyzing the robustness and sensitivity of the design. Because we selected an example circuit from the Modelica Standard Library (MSL), the initial design structure (i.e., seed design) was given. Even for this simple example most of the time is spent on creating the component library ($\sim$ 19%) and creating the design space with constraints ($\sim$ 19%).

If this design task were performed using Modelica alone, some of the steps we took would be identical (e.g., creating a requirement specification, creating a component library). However, other steps (e.g., exploring the discrete design space manually) would have taken far more time, depending on the scale or the complexity of the design. For instance, handling complex design constraints. Modelica supports replaceable packages, which can be used to ensure that the same component instance is selected for the transistor, the resistor, and the capacitor pairs, but the constraint on R_BC for timing cannot be expressed by using the Modelica language. Therefore, the configurations in the design space must be manually created, assuming no other external tools are used (such a tool may need to be developed specifically for the task). Using a manual process it takes approximately 15 min to set up one configuration in Modelica, execute the simulation, and extract and organize the results. For 84 configurations, it means $84 \times 15\text{min} = 1260\text{min} = 21\text{h}$. OpenMDAO [31] is used to define the PCC and DoE analyses. Assuming an OpenMDAO wrapper exists for Modelica, to set up and run all analyses takes 3 h using parallel execution. By adding the individual activities together $30\text{min} + 2\text{h} + 84 \times 15\text{h} + 3\text{ h} = 26.5\text{h}$ is required to design and analyze the same system, which is $\sim$ 2.47 times slower than our design process.

However, if there is only a single design with no alternative choices and configurations then the total time is $30\text{min}+1\times15\text{min}+30\text{ min} = 1.25\text{h}$ (no component library and only one configuration is considered). In this case, our design process is $\sim$ 8.29 times slower than the traditional design process. In both cases, the initial design is given and it would add the same constant time to both processes. We only evaluated and analyzed the additional activity, which is significantly different in both processes. Our experiment shows that the traditional design process is preferred and more effective if there are: (a) one or a few design variations considered, (b) one or a few alternative components considered without any constraints, and (c) a single analysis tool is used. However, we can conclude that our Analysis-driven

Activity	**Required time**
Created a requirement specification	30min
Built a component library for transistors	2h
Imported the components into CyPhy	15min
Created resistor and capacitor components in CyPhy	30min
Created the design in CyPhy	15min
Imported the test components in CyPhy	5min
Created the Test Bench in CyPhy	35min
Debuged the generated model to get it working	1h
Implemented post processing scripts	1h
Created a design space with constraints ($90,000/84$)	2h
Executed 12 designs, one failed, changed solver settings	2min
Executed 84 designs, two failed, changed solver settings	10min
Analyzed the results, set metric weights	10min
Created a PCC model for the design space	30min
Created a DoE model for the design space	30min
Executed PCC over the entire design space 84 cfgs	45min
Executed DoE for 8 cfgs	5min
Total time	**10**h **22**min
Design time ($\sim$ 88.42%)	9h 10min
Execution time ($\sim$ 11.58%)	1h 12min

Table 8: Oscillator: Design time

Rapid Design Process (ARDP) is preferred and more effective if there are: (a) several (e.g., tens or hundreds) of possible design variations, (b) several alternative components (i.e., a large component library) are considered with design constraints, and (c) multiple (possibly interdependent) analysis tools are used.

Ground vehicle driveline

Our case study is the design of a ground vehicle driveline, subject to a set of system-level requirements. The following sections walk through the stages of an Analysis-driven Rapid Design Process (ARDP) for this driveline example to find the best possible design point (i.e., prototype) with respect to the defined requirements. This case study does not explore alternative architectures (e.g., hybrid driveline), but our example can easily be extended to include such alternatives, because the tools used herein already provide support for representing alternative design architectures. Alternative architectures would consider topologically different designs, where the variations are not only in the component choices, but also in the interactions between components.

Requirements

The driveline model has 19 automotive performance requirements that are divided into five subcategories: *Speed*, *Acceleration*, *Range*, *Temperature/Cooling*, and *Fuel Economy*. To illustrate the Analysis-driven Rapid Design Process, we consider 11 of the 19 requirements. The *Speed* requirement category contains five maximum speed requirements (forward speed, hill climb on different surfaces, and reverse speed) and two average speed requirements, one using a drive cycle defined from the US06 Supplemental FTP Driving Schedule [3].

Each requirement has a threshold and an objective value. The threshold is the pass/fail mark, and the objective represents the ideal outcome. A score can be assigned to each metric by comparing it to the threshold and objective; exceeding the objective will not necessarily have increased benefit. A design's overall (weighted) score is computed based on the requirement structure and analysis results, and represents the quality of the design, which facilitates the quantitative comparison of different design variations.

Analysis templates

Analysis templates, called CyPhy *Test Benches* in the OpenMETA tools [13, 69], are the executable versions of the requirements. Figure 31 depicts a CyPhy *Test Bench* model containing a driveline design, test components, environmental conditions, driver profile, four parameters (i.e., inputs), and five metrics (i.e., outputs). This *Test Bench* evaluates five of the requirements for a driveline design point. In order to evaluate other requirements, additional Test Benches are implemented in the OpenMETA tools; the collection of *Test Benches* that captures the entire set of requirements is defined as a *Set of Test Benches* (SoT). By executing the SoT model, all requirements are evaluated for a single point design.

Single Design Point (architecture seed design)

A *seed design* is built as a *Component Assembly* in CyPhy, and it is used to hash out the initial component choices and the general architecture of the design. The architecture consists of a cooling system, software controllers (Engine Control Unit and Transmission Control Unit), left-hand and right-hand side drive (each includes a drive shaft and a final

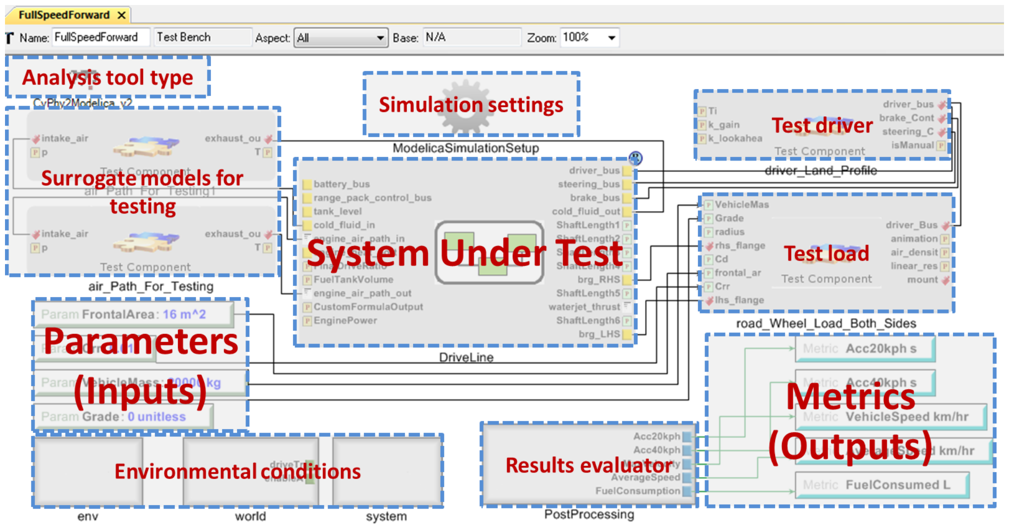


Figure 31: Vehicle driveline: Full Speed Forward Test Bench [83]

drive), two surrogate fluid models (air path and fluid sink), a battery, a fuel tank, an engine, and a transmission. The Engine Control Unit (ECU) and Transmission Control Unit (TCU) software *Components* are implemented using the Cyber Composition Language [109] and corresponding C++ code is generated for simulation purposes.

The key components from the *Speed* and *Acceleration* requirements category point of view are the engine and the transmission components. We focus only on these two key components to improve the performance characteristics of the driveline design in order to meet all requirements. The seed design has a Deutz BFM1015M (290HP) engine and an Allison X200 4A (4 forward gears) transmission. It is also possible to consider different engine and transmission alternatives by using the OpenMETA tools and programmatically turning the seed design (i.e., single design point) into a discrete design space.

Discrete Design Space

The discrete design space resulting from the single design point has the exact same hierarchical decomposition structure. The discrete design space for the driveline model shows that for the engine and the transmission, designers can consider alternative components. In other words, different engine and transmission component models can be added and all possible design points are encoded by this design space. Table 9 and Table 10 summarize some of the engine and transmission options, respectively, and their important parameters that we considered based on publicly available data sheets from the manufacturers' websites [2, 28, 66, 67, 142]. The highlighted transmission and engine instances were used in the original seed design.

Supplier	**Type**	**HP**
Caterpillar	C9 280kW	375
Caterpillar	C11 313kW	420
Caterpillar	C15 444kW	595
Caterpillar	C18 597kW	800
Caterpillar	C27 597kW	800
Caterpillar	C32 709kW	950
MTU	MT883 644kW	864
MTU	6V199 261kW	350
MTU	6V199 335kW	455
MTU	6V199 430kW	585
MTU	8V199 530kW	720
MTU	8V199 603kW	820
MTU	8VMT881 736kW	1000
MTU	12VMT883 1103kW	1500
MTU	6R106Euro3 240kW	325
Deutz	**BF6M1015M group A**	**290**
Deutz	BF6M1015MC group A	385
Deutz	TCD2015V6M group A	440
Deutz	TCD2015V8M group A	600
Cummins	QSM 350HP FR20019	350
Cummins	QSM 400HP FR20003	400
Cummins	QSX 400HP FR10581	400
Cummins	QSX 500HP FR10583	500

Table 9: Vehicle driveline: Engine alternatives (23/25) [83]

The discrete design space contains 25 engine alternatives and 8 transmission alternatives yielding 200 configurations. Clearly, adding alternative components for every design container will quickly explode the number of possible configurations. As a direct result of the large number of configurations, the time required for executing all Test Benches (8 in this example, but there could be many more) over the entire design space becomes computationally expensive and thereby increases the design time. Elaborating all possible configurations is not scalable and is unnecessary, because there are many engine/transmission combinations that cannot be physically realized. Using design space constraints (symbolic expressions over

Supplier	Type	Minimum HP	Maximum HP	# of forward gears	# of reverse gears
Allison	**X200-4A**	**206.36**	**344.38**	**4**	**1**
Allison	X200-4B	239.86	399.32	4	2
Allison	XT1410-4	473.02	787.92	3	1
Allison	XT1410-5A	473.02	787.92	3	1
Allison	XTG411-2A	213.06	355.1	4	2
Allison	XTG411-4	340.36	566.82	4	2

Table 10: Vehicle driveline: Transmission alternatives (6/8) [83]

the design variables that must be true for all feasible designs) we can prune the combinatorial design space to a smaller set of designs which allows us to elaborate and analyze only the viable configurations. Rapidly eliminating non-viable configurations is a powerful technique, and saves precious time and computational resources, which facilitates our rapid design process. For the driveline model we used two property constraints: the engine output power rating must be within the range of the transmission's minimum and maximum input power rating. For instance, according to the data sheets, Caterpillar C15 444kW engine from Table 9 can only be used with one of two transmissions: Allison XT1410-4 and XT1410-5A from Table 10, because the engine's output power is 595 HP, which exceeds all other transmissions' maximum input power rating.

The next step of this design process is to evaluate all requirements for every *viable* generated design point. Eight CyPhy *Test Benches* already exist for the original seed design. The OpenMETA tools support the reuse of these *Test Benches* for an entire design space by changing the top level system under test object in every *Test Bench* to point to the newly created *Design Space* container. This approach produces reusable analysis templates (*Test Bench* models) for any design point within the discrete design space as long as the system under test component has the same interface. A *Set of Test Benches* can also be used with a discrete design space to execute all *Test Benches* from a single entry-point.

Once all *Test Bench* results are available, the Project Analyzer helps designers to understand which designs meet the requirements and which ones do not (and by how much). One of the visualization techniques is the parallel axis plot as shown on Figure 32, where each design is represented by a single line and each line is color-coded. The color-coding is either based on *Component*-level limit violations (see Figure 32), or whether the requirements are met or not, or ranking. On each vertical axis the threshold values (red) and the objective values (green) are marked for each metric.

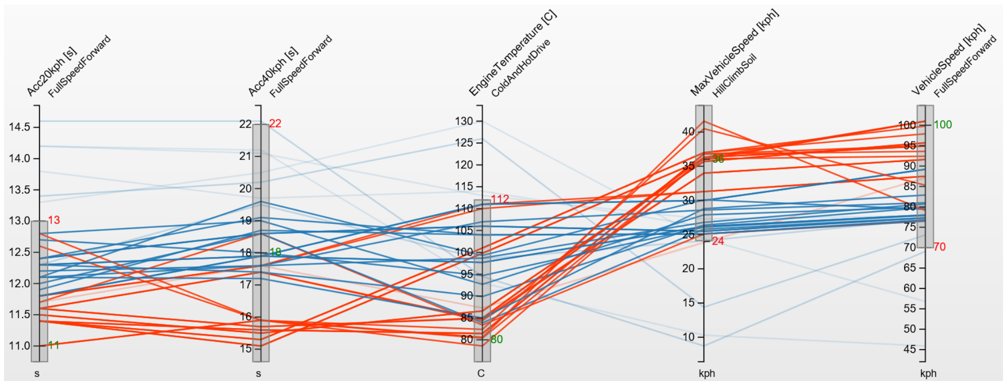


Figure 32: Vehicle driveline: Physical limit violations [83]

All of these capabilities guide the designers to make decisions about which design points should be considered for further and more detailed analyses. In our use case we selected configurations 2, 4, 7, 30, and 43 for parametric design exploration. Note: the original seed design is configuration 41. Performing these analyses during the design process, before all design details have been finalized, facilitates the Analysis-driven Rapid Design Process.

Parametric Design Space

In the previous section the discrete design space exploration was presented in the context of the driveline model. The OpenMETA tools support parametric design space exploration through the *Parametric Exploration* models. In most cases, parametric analyses take significantly more time than discrete, because all *Test Benches* are executed multiple times for a single design point. These parametric analyses are used to assess the robustness of the designs and to generate surrogate models for the *Test Benches*. For instance, even if a design meets all requirements using the mean values of component parameters and environment conditions, it is important to see the impact on the metric values (i.e., output of the *Test Benches*) if parameters (i.e., inputs of the *Test Benches*) have variations or if they can change within a certain range during normal operating conditions.

System Robustness Analysis

A *Probabilistic Certificate of Correctness (PCC)* [61] parametric driver is set up for the Full Speed Forward *Test Bench* as shown in Figure 33. The grade and the mass parameters are defined as probability density functions (pdf), where the grade has a uniform distribution between a minimum and a maximum value and the mass has a normal distribution with a mean and a variance. Generated results show the impact on the outputs (acceleration to $20\frac{\text{km}}{\text{h}}$, average speed, and vehicle speed) of the *Test Bench*. The minimum values and the

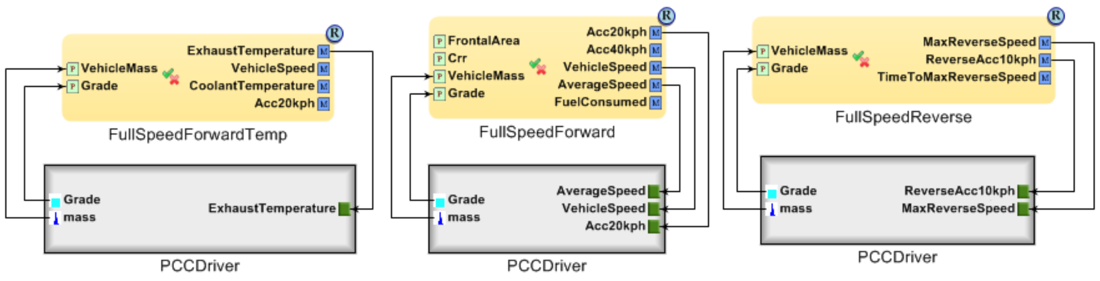


Figure 33: Vehicle driveline: Parametric Design Space Exploration Models [83]

acceptable PCC target values are defined in the parametric exploration model by the user and they are visualized on the plots by the Project Analyzer. The PCC value for each output is the area below the output pdfs within the minimum and maximum range.

PCC experiments can be defined for multiple *Test Benches* as shown in Figure 33, which means a system robustness analysis is performed for multiple requirements. For each *Test Bench* one "joint" PCC value is computed based on the individual PCC values for each metric [61]. The joint PCC is always a real number with a value between 0 and 1.

The Parametric Design Exploration using PCC drivers could be applied for an entire discrete design space, but these analyses would consume valuable resources to (unnecessarily) perform in-depth analyses on design configurations which do not satisfy design requirements. Based on initial high-level system analyses, we have chosen only three *Test Benches* and five promising configurations (2, 4, 7, 30, and 43) to further analyze using PCC. The Project Analyzer visualizes the PCC results for multiple *Test Benches* and configurations in the form of a heat map. Figure 34 depicts the robustness of each design point with respect to each Test Bench. Based on this figure, configuration 30 has the highest accumulated PCC value across all three selected Test Benches, therefore configuration 30 is the most robust design in this set.

Surrogate Model and Prediction Profiler

The OpenMETA tools [69] support surrogate model generation from a parametric exploration model that contains a parameter study (i.e., Design of Experiment (DoE)) driver. Generating surrogate models is computationally intensive, but after a surrogate model is generated, interactive 2D and 3D plots can be generated for the users to predict the metrics within the parametric space where the surrogate model is valid.

Configuration 30 was selected based on the system robustness analysis (i.e., the PCC experiments). For configuration 30, a surrogate model was created by the OpenMETA tools. The resulting surrogate model can be saved in two different forms: (a) an exported binary

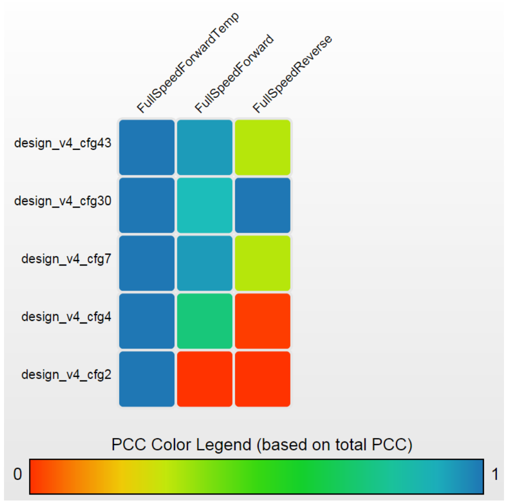


Figure 34: Vehicle driveline: Results of Parametric Exploration over a discrete design space [83]

Python pickle object that can be loaded and analyzed later and (b) a set of polynomial equations that are used to approximate *Test Bench* outputs. In the Project Analyzer, parameters can be selected and metrics are shown with respect to parameter variations within the validity of the surrogate model. Furthermore, the prediction profiler estimates both the $0 \rightarrow 20\frac{\mathrm{km}}{\mathrm{h}}$ acceleration time and the maximum vehicle speed based on the selected input parameters (e.g., the coefficient of rolling resistance (C_{rr}) and the frontal area of the vehicle). For each pair of input parameters, a 3D response surface is plotted where metrics (e.g., $0 \rightarrow 40\frac{\mathrm{km}}{\mathrm{h}}$ acceleration time) are shown as 3D surfaces w.r.t. the inputs (e.g., the coefficient of rolling resistance and the frontal area of the vehicle). Designers can choose which parameters (inputs) and metrics (outputs) are plotted using the Project Analyzer based on the available variables from the defined *Test Bench* analysis templates.

Results

The original driveline design point (seed design) does not meet the requirements as shown in Table 11. This driveline seed design is turned into a design space and alternative engine and transmission components are added. The discrete design space generates 200 possible configurations, and after constraints are implemented 67 viable configurations remain in the pruned design space.

The 67 viable configurations are analyzed to evaluate all requirements across the discrete design space. Many design points meet all requirements, and 5 of them are selected for further parametric design space analysis to determine system robustness. Finally, configuration 30 is chosen to generate a surrogate model for the driveline design problem. The surrogate model is used to predict the key performance parameters of the system under varying (environmental)

Metric	**Threshold**	**Seed design (cfg. 41)**	**Final design (cfg. 30)**
Speed Category	$\frac{km}{h}$	$\frac{km}{h}$	$\frac{km}{h}$
Max Full Speed Forward	min 70	69.1 (F)	76.9 (P)
Max Hill Climb Soil	min 24	8.72 (F)	24.1 (P)
Max Hill Climb Sand	min 10	0 (F)	13.3 (P)
Max Hill Climb Concrete	min 30	30.2 (P)	37.1 (P)
Avg Speed Highway	min 40	57.8 (P)	61.4 (P)
Avg Speed Forward	min 40	37.9 (F)	43.8 (P)
Max Speed Reverse	min 19	20 (P)	21.5 (P)
Acceleration Category	s	s	s
$0 \rightarrow 20\frac{km}{h}$	max 13	14.6 (F)	12.3 (P)
$0 \rightarrow 40\frac{km}{h}$	max 22	22.1 (F)	19.5 (P)
Reverse $0 \rightarrow 10\frac{km}{h}$	max 8	8.7 (F)	7.62 (P)
Hill Climb $0 \rightarrow 20\frac{km}{h}$	max 80	81.2 (F)	74.2 (P)

Table 11: Vehicle driveline: Requirements for design problem [83]

conditions. Table 11 shows that all requirements are met for the final driveline design (configuration 30).

Additional case studies are presented [13, 15] including vehicle design challenges and an excavator model.

Potential Other Applications

Our approach to multi-domain system design can be applied to a wide range of problems, notably the design of electronic devices such as modular smartphones [14]. Smartphone design must address cost, weight, spatial, performance, power, software, and thermal requirements, among others. A single analysis tool *cannot* easily evaluate, track, and balance all these factors; our rapid process can leverage multiple existing tools and to efficiently produce an architectural prototype. As new and improved *Components* (e.g., batteries, image sensors, wireless communication modules, etc.) become available, the effect on the overall design's performance can easily be assessed.

Our Analysis-driven Rapid Design Process (ARDP) could be used for Cyber-Physical System (CPS) component library development. Analysis template models are created for component unit testing; for every change a set of analyses is automatically executed for all component instances. This could potentially shorten the development time of CPS component libraries. Component libraries often contain parametric component models, for instance a parametric resistor model. As new component instances become available, the existing component library can be tested with all new instance parameters to verify that the parametric component models support the new instances. In addition to this unit testing, the component models from the library could be used to create one or more system models, and automated integration testing could be performed in several design contexts. As experimental data becomes available automated validation of the component library models can be created in the form of analysis template models.

Another possible application is stress testing the component library and the analysis templates within the expected parametric ranges. The expected parametric ranges for the components and analysis templates could be defined in parametric exploration models. By specifying a parametric exploration driver component (e.g., DoE or PCC) simulation or analysis setup issues can be easily discovered in the early stages of the design process as we presented it in one of our use case (oscillator design).

Finally, product lines can be often modeled with a discrete design space and discrete choices of components and topologies. The benefit of utilizing the discrete design space capabilities are: (a) constraints can be grouped and applied for each product line, (b) a single model can hold all viable configurations (i.e., product line variants), and (c) all analysis templates can be reused for all product lines.

CHAPTER VII

CONCLUSION

The Defense Advanced Research Projects Agency (DARPA) created an Adaptive Vehicle Make (AVM) program portfolio, which defined several projects to reduce the design and manufacturing time and to democratize the engineering design processes. The overall topic is necessarily broad because it includes: design, verification, validation, and manufacturing. To address all of these topics, the projects required expertise from different universities, research institutions, and companies. It was a collaborative effort to work on the individual projects and integrate the outcomes into the AVM portfolio. In these projects, it is difficult to pinpoint individual contributions. We made significant contributions to the topics summarized below.

Contribution 1 Designed and implemented the dynamics aspect of the CyPhyML model integration language, which supports heterogeneous component models [27, 84, 85, 106]. Defined the dynamics concepts for the AVM component model specification to support model interoperability between tools [105].

Contribution 2 Defined executable analysis template models to evaluate requirements, composition of interdependent analysis template models, and parametric design space exploration models to perform system robustness studies [60, 86, 132].

Contribution 3 Implemented model composers for several analysis tools including Modelica and OpenMDAO [82, 107].

Contribution 4 Defined four integration platforms, the conceptualization of the integration architecture, and implemented prototypes of the key concepts and functions of the integration architecture [73, 81].

Contribution 5 Defined and evaluated an Analysis-driven Rapid Design Process (ARDP) for Cyber-Physical Systems (CPSs) [83].

Lessons learned

Product design has become increasingly complex in recent decades. Several design methods and processes have been developed to reduce the design complexity to a manageable level. These methods and approaches include: layered design, component-based design, the v-model, model-based development, virtual integration, platform-based design, and contract-based design. Model-based systems engineering leverages these methods to manage design complexity and to reduce development time and costs. In contrast to the traditional document-centric design process, model-based design approaches maintain traceability and dependency among design artifacts in the form of relationships between models. The building blocks of a model-based engineering tool are models that represent the requirements, system components, and subsystems. Using model-based systems engineering tools has become an accepted practice.

Most of these design methods and processes were primarily developed for either software or hardware development. However, in Cyber-Physical Systems (CPSs), computational elements are tightly integrated with physical processes and physical components. Computational elements often interact with the physical system through a distributed network. Physical systems are acausal systems, which means that they do not have predefined inputs and outputs by nature. For a specific operating mode of the system, inputs and outputs can be derived, but it is a cumbersome process for a large complex system.

There are hundreds of distinct tools used in the automotive and aerospace industry to analyze different aspects of a complex system design. These tools include both in-house, i.e., internally developed and maintained, (70%) and commercial off-the-self tools (30%), which shows that there is no single tool that can deal with all aspects of a complex design problem. An adequate model of Cyber-Physical Systems must: (1) capture domain interactions (e.g., electrical power and mechanical systems), (2) incorporate multiple aspects and domain models for each component, (3) support a wide variety of analysis techniques, (4) enable the reuse of existing models from libraries, and (5) extract sufficient information from model libraries to support architecture exploration for product families.

Results

We presented the key concepts and functions of an Analysis-driven Rapid Design Process (ARDP) for Cyber-Physical Systems (CPSs) to address all aforementioned challenges. We defined and contributed to three platforms to improve efficiency and quality of the design process: a Model Integration Platform, Tool Integration Platform, and Execution Integration Platform. The Model Integration Platform (a) uses heterogeneous component models, (b) keeps the multi-domain models consistent, (c) tracks model dependencies, and (d) facilitates importing from existing model libraries. The Tool Integration Platform accommodates a variety of analysis tools with the flexibility to add new tools in the future. The Execution

Integration Platform provides an analysis tool independent framework for analysis execution and organization of analysis results. In addition to several platform contributions, we proposed a design flow to achieve an analysis-driven rapid iterative design process.

Finally, we demonstrated and evaluated this design flow utilizing two case studies: (a) an oscillator circuit design and (b) a ground vehicle driveline design. In both cases, we assumed that component model libraries exist and the models are composable. We compared our design process to the traditional design process for the oscillator circuit design. The result of that comparison showed us that our design process can be faster or slower depending on the product to be designed. When we used a single analysis tool (e.g., Modelica) and a single design was considered without any component or topology variations our design process was ~ 8.29 times slower than the traditional design process. However, if we considered several component alternatives and used design constraints our design process was ~ 2.47 times faster than the traditional design process with 84 configurations in the design space. Designers can benefit from our Analysis-driven Rapid Design Process (ARDP) when: (a) several component choices are available, (b) design families or configurable product lines are considered, (c) multiple interdependent analysis tools are used to evaluate requirements, and (d) multiple domain models must be kept consistent.

Open research challenges

Our work leads to several other research challenges: (1) how to capture the relationship between the discrete and parametric design space, (2) how to support multi-fidelity models on the component-level and subsystem-level, (3) how to guide designers with the selection of multi-fidelity models for components and subsystems, (4) how to address the hierarchical decomposition if each domain requires a different system hierarchy, and (5) how to incorporate product life cycle management tools to this design process? Addressing these research challenges could significantly improve the effectiveness of our Analysis-driven Rapid Design Process (ARDP).

APPENDIX A

RELEVANT PUBLICATIONS

Heterogeneous component models

1. Adam Nagel, Sandeep Neema, Mike Myers, Robert Owens, Zsolt Lattmann, and Dan Finke. AVM Component Specification version 2.5. `https://github.com/metamorph-inc/meta-core/blob/master/meta/DesignDataPackage/doc/AVM_Component_Spec.pdf`, 2014. last accessed: 04/04/2016

2. Himanshu Neema, Jesse Gohl, Zsolt Lattmann, Janos Sztipanovits, Gabor Karsai, Sandeep Neema, Ted Bapty, John Batteh, Hubertus Tummescheit, and Chandrasekar Sureshkumar. Model-based integration platform for fmi co-simulation and heterogeneous simulations of cyber-physical systems. In *Proceedings of the 10th International Modelica Conference*, pages 235–245, Lund University, olvegatan 20A, SE-223 62 LUND, SWEDEN, 03 2014. Modelica Association and Linoping University Electronic Press

3. Joshua D Carl, Zsolt Lattmann, and Gautam Biswas. Modeling and simulation semantics for building large-scale multi-domain embedded systems. In *27th European Conference on Modelling and Simulation*, Norway, 05 2013

4. Zsolt Lattmann, Adam Nagel, Tihamer Levendovszky, Ted Bapty, Sandeep Neema, and Gabor Karsai. Component-based modeling of dynamic systems using heterogeneous composition. In *6th International Workshop on Multi-Paradigm Modeling*, Innsbruck, Austria, 10 2012

5. Zsolt Lattmann, Adam Nagel, Jason Scott, Kevin Smyth, Johanna Ceisel, Chris vanBuskirk, Joseph Porter, Sandeep Neema, Ted Bapty, Dimitri Mavris, and Janos Sztipanovits. Towards automated evaluation of vehicle dynamics in system-level designs. In *Proc. ASME International Design Engineering Technical Conf. & Computers and Information in Engineering Conf. (IDETC/CIE 2012)*, Chicago, IL, USA, 08 2012

Analysis templates and model execution framework

1. Zsolt Lattmann, James Klingler, Patrik Meijer, Ted Bapty, Sandeep Neema, and Jason Scott. Integration platform technology components in the meta toolchain. Technical Report ISIS-15-110, Institute for Software Integrated Systems, Nashville, 01/2015 2015

2. Tomonori Honda, Eric Saund, Ion Matei, William Janssen, Bhaskar Saha, Daniel G. Bobrow, Johan de Kleer, Tolga Kurtoglu, and Zsolt Lattmann. A simulation and modeling based reliability requirement assessment methodology. In *ASME 2014 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Buffalo, NY, USA, 08 2014

3. Bhaskar Saha, Tomonori Honda, Ion Matei, Eric Saund, Johan de Kleer, Tolga Kurtoglu, and Zsolt Lattmann. A model-based approach for an optimal maintenance strategy. In *2nd European Conference of the PHM Society - PHME'14*, Nantes, France, 07 2014

4. Zsolt Lattmann, Adrian Pop, Johan de Kleer, Peter Fritzson, Bill Janssen, Sandeep Neema, Ted Bapty, Xenofon Koutsoukos, Matthew Klenk, Daniel Bobrow, Bhaskar Saha, and Tolga Kurtoglu. Verification and design exploration through meta tool integration with openmodelica. In *Proceedings of the 10th International Modelica Conference*, pages 353–362, Lund University, olvegatan 20A, SE-223 62 LUND, SWEDEN, 03 2014. Modelica Association and Linoping University Electronic Press

5. Laszlo Juracz, Zsolt Lattmann, Tihamer Levendovszky, Graham Hemingway, Will Gaggioli, Tanner Netterville, Gabor Pap, Kevin Smyth, and Larry Howard. Vehicleforge: A cloud-based infrastructure for collaborative model-based design. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud computing co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, volume Vol-1118, 01 2014

Analysis-driven Rapid Design Process

1. Zsolt Lattmann, James Klingler, Patrik Meijer, Jason Scott, Sandeep Neema, Ted Bapty, and Gábor Karsai. Towards an Analysis-Driven Rapid Design Process for Cyber-Physical Systems. *26th IEEE International Symposium on Rapid System Prototyping (RSP)*, 10 2015

2. Zsolt Lattmann, James Klingler, Patrik Meijer, Ted Bapty, Sandeep Neema, and Jason Scott. Meta design space exploration using dynamics. Technical Report ISIS-15-106, Institute for Software Integrated Systems, Nashville, 01/2015 2015

3. Himanshu Neema, Zsolt Lattmann, Patrik Meijer, James Klingler, Sandeep Neema, Ted Bapty, Janos Sztipanovits, and Gabor Karsai. Design space exploration and manipulation for cyber physical systems. In *IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems (IDEAL' 2014)*, Berlin, Germany, 04 2014. Springer, Springer

APPENDIX B

OTHER PUBLICATIONS

1. Siyuan Dai, Zsolt Lattmann, and Xenofon Koutsoukos. Compositional control design of cyber-physical systems. In Danda B Rawat, Joel Rodrigues, and Ivan Stojmenovic, editors, *Cyber Physical Systems: From Theory to Practice*. CRC Press, 2015

2. Peng Zhang, Zsolt Lattmann, James Klingler, Sandeep Neema, and Ted Bapty. Visualization techniques in collaborative domain-specific modeling environment. In *SoutheastCon 2015*, Fort Lauderdale, FL, USA, 04/2015 2015. IEEE

3. Ted Bapty, Justin Knight, Zsolt Lattmann, Sandeep Neema, and Jason Scott. Software quality assurance for the meta toolchain. Technical Report ISIS-15-111, Institute for Software Integrated Systems, Nashville, 01/2015 2015

4. Joseph Porter, Zsolt Lattmann, Graham Hemingway, Nagabhushan Mahadevan, Sandeep Neema, Harmon Nine, Nicholas Kottenstette, Peter Volgyesi, Gabor Karsai, and Janos Sztipanovits. The esmol modeling language and tools for synthesizing and simulating real-time embedded systems. In *15th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, CA, 04 2009

APPENDIX C

SIMULATION ERROR LOG FOR OSCILLATOR DESIGN

Oscillator design example error message generated by the DASSL solver using Dymola.

```
... "dsin.mat" loading (dymosim input file)
... "Oscillator_Design_cfg9.mat" creating (simulation result file)
Integration started at T = 0 using integration method DASSL
(DAE multi-step solver (dassl/dasslrt of Petzold modified
by Dynasim))
Integration terminated before reaching "StopTime" at T = 0.0338
   CPU-time for integration       : 0.102 seconds
   CPU-time for one GRID interval: 0.0362 milli-seconds
   Number of result points        : 2817
   Number of GRID   points        : 2817
   Number of (successful) steps   : 4235
   Number of F-evaluations        : 19860
   Number of Jacobian-evaluations: 1427
   Number of (model) time events : 2
   Number of (U) time events      : 0
   Number of state    events      : 0
   Number of step     events      : 0
   Minimum integration stepsize   : 2.24e-011
   Maximum integration stepsize   : 0.000197
   Maximum integration order      : 5
Calling terminal section
... "dsfinal.txt" creating (final states)

... Error message from dymosim.exe
At current time T = 3.378006e-002 the local error test cannot be
satisfied either because a solution is infinite (i.e. unstable
system), or because a component of TolAbs is zero and the
computed solution is zero too. Integration is terminated.
```

APPENDIX D

DESIGN SPACE RESULTS FOR OSCILLATOR

	Id	**Transistor**	R_B	C	**P/F**	f	t_r	I_{min}	Score
1	cfg39	2SC5200	27kΩ	0.082µF	Pass O.	471Hz	180µs	0.969µA	95.23
2	cfg72	BC847A	24kΩ	0.1µF	Pass O.	508Hz	216µs	1.11µA	92.58
3	cfg8	2N5551	24kΩ	0.1µF	Pass O.	471Hz	216µs	1.22µA	91.27
4	cfg54	2N4401	24kΩ	0.1µF	Pass O.	502Hz	204µs	1.74µA	90.71
5	cfg45	BC547A	27kΩ	0.082µF	Pass T.	343Hz	180µs	1.16µA	89.55
6	cfg30	2N3904	24kΩ	0.1µF	Pass O.	499Hz	204µs	2µA	89.30
7	cfg3	BC550C	27kΩ	0.082µF	Pass T.	327Hz	204µs	0.465µA	89.21
8	cfg18	2N3055	27kΩ	0.082µF	Pass T.	425Hz	180µs	1.98µA	89.04
9	cfg73	BC847A	27kΩ	0.082µF	Pass T.	326Hz	180µs	1.11µA	88.99
10	cfg27	2SC4793	27kΩ	0.1µF	Pass O.	505Hz	204µs	2.15µA	88.51
11	cfg11	2N5551	27kΩ	0.082µF	Fail	324Hz	180µs	1.33µA	88.32
12	cfg41	2SC5200	27kΩ	0.1µF	Pass T.	329Hz	216µs	0.972µA	88.03
13	cfg76	BC847A	20kΩ	0.12µF	Pass T.	490Hz	264µs	1.11µA	87.45
14	cfg37	2SC5200	24kΩ	0.1µF	Fail	306Hz	216µs	0.985µA	87.17
15	cfg38	2SC5200	22kΩ	0.12µF	Pass T.	379Hz	252µs	0.969µA	86.99
16	cfg43	BC547A	24kΩ	0.1µF	Fail	304Hz	216µs	1.17µA	86.71
17	cfg52	2N4401	20kΩ	0.12µF	Pass T.	484Hz	252µs	1.75µA	86.66
18	cfg12	2N5551	20kΩ	0.12µF	Pass T.	455Hz	264µs	1.33µA	86.54
19	cfg2	BC550C	24kΩ	0.1µF	Fail	301Hz	240µs	0.460µA	86.27
20	cfg32	2N3904	27kΩ	0.082µF	Pass T.	326Hz	168µs	1.99µA	86.19
21	cfg74	BC847A	27kΩ	0.1µF	Fail	267Hz	204µs	1.11µA	85.98
22	cfg36	2N3904	20kΩ	0.12µF	Pass T.	462Hz	252µs	2.01µA	85.62
23	cfg46	BC547A	27kΩ	0.1µF	Fail	271Hz	216µs	1.15µA	85.57
24	cfg22	2SC4793	27kΩ	0.082µF	Pass T.	329Hz	180µs	2.15µA	85.49
25	cfg6	BC550C	27kΩ	0.1µF	Fail	268Hz	240µs	0.459µA	85.20
26	cfg10	2N5551	27kΩ	0.1µF	Fail	266Hz	216µs	1.33µA	84.98
27	cfg23	2SC4793	24kΩ	0.1µF	Pass T.	331Hz	204µs	2.17µA	84.82
28	cfg79	2N2222	24kΩ	0.1µF	Fail	303Hz	216µs	1.86µA	84.82
29	cfg48	BC547A	20kΩ	0.12µF	Fail	303Hz	252µs	1.15µA	84.56

	Id	**Transistor**	R_B	C	**P/F**	f	t_r	I_{min}	Score
30	cfg19	2N3055	27kΩ	0.1µF	Fail	305Hz	216µs	1.97µA	84.34
31	cfg20	2N3055	24kΩ	0.1µF	Fail	305Hz	216µs	1.98µA	84.30
32	cfg50	BC547A	22kΩ	0.12µF	Fail	288Hz	252µs	1.15µA	84.11
33	cfg42	2SC5200	20kΩ	0.12µF	Fail	305Hz	264µs	0.971µA	84.10
34	cfg5	BC550C	20kΩ	0.12µF	Fail	302Hz	276µs	0.464µA	83.77
35	cfg75	BC847A	22kΩ	0.12µF	Fail	272Hz	252µs	1.11µA	83.71
36	cfg80	2N2222	27kΩ	0.1µF	Fail	270Hz	216µs	1.85µA	83.64
37	cfg16	2N3055	22kΩ	0.12µF	Pass	346Hz	252µs	1.97µA	83.41
38	cfg13	2N5551	22kΩ	0.12µF	Fail	271Hz	252µs	1.34µA	83.20
39	cfg44	BC547A	24kΩ	0.12µF	Fail	253Hz	252µs	1.15µA	83.06
40	cfg71	BC847A	24kΩ	0.12µF	Fail	250Hz	252µs	1.11µA	83.04
41	cfg33	2N3904	27kΩ	0.1µF	Fail	266Hz	216µs	1.99µA	83.03
42	cfg35	2SC5200	24kΩ	0.12µF	Fail	255Hz	264µs	0.968µA	82.70
43	cfg84	2N2222	20kΩ	0.12µF	Fail	303Hz	252µs	1.87µA	82.65
44	cfg9	2N5551	24kΩ	0.12µF	Fail	250Hz	252µs	1.33µA	82.59
45	cfg17	2N3055	20kΩ	0.12µF	Fail	304Hz	252µs	1.98µA	82.32
46	cfg4	BC550C	22kΩ	0.12µF	Fail	274Hz	288µs	0.46µA	82.21
47	cfg83	2N2222	22kΩ	0.12µF	Fail	286Hz	252µs	1.87µA	82.19
48	cfg26	2SC4793	20kΩ	0.12µF	Fail	323Hz	252µs	2.19µA	82.05
49	cfg1	BC550C	24kΩ	0.12µF	Fail	251Hz	288µs	0.459µA	81.65
50	cfg47	2N4401	24kΩ	0.12µF	Fail	250Hz	252µs	1.74µA	81.56
51	cfg21	2SC4793	24kΩ	0.12µF	Fail	298Hz	252µs	2.17µA	81.50
52	cfg29	2N3904	22kΩ	0.12µF	Fail	272Hz	252µs	2.01µA	81.36
53	cfg78	2N2222	24kΩ	0.12µF	Fail	252Hz	252µs	1.88µA	81.27
54	cfg15	2N3055	24kΩ	0.12µF	Fail	255Hz	252µs	1.97µA	81.01
55	cfg25	2SC4793	22kΩ	0.12µF	Fail	276Hz	252µs	2.18µA	80.89
56	cfg31	2N3904	24kΩ	0.12µF	Fail	250Hz	252µs	1.99µA	80.81
57	cfg40	2SC5200	18kΩ	0.15µF	Fail	311Hz	312µs	0.971µA	80.56
58	cfg28	2SC4793	18kΩ	0.15µF	Fail	468Hz	312µs	2.2µA	79.68
59	cfg49	BC547A	18kΩ	0.15µF	Fail	270Hz	312µs	1.14µA	79.44
60	cfg77	BC847A	18kΩ	0.15µF	Fail	265Hz	312µs	1.11µA	79.38
61	cfg14	2N5551	18kΩ	0.15µF	Fail	265Hz	312µs	1.36µA	78.90
62	cfg56	2N4401	18kΩ	0.15µF	Fail	265Hz	312µs	1.76µA	77.97
63	cfg24	2N3055	18kΩ	0.15µF	Fail	287Hz	312µs	1.97µA	77.85
64	cfg7	BC550C	18kΩ	0.15µF	Fail	268Hz	348µs	0.462µA	77.77

	Id	**Transistor**	R_B	C	**P/F**	f	t_r	I_{min}	Score
65	cfg82	2N2222	18kΩ	0.15µF	Fail	268Hz	312µs	1.88µA	77.72
66	cfg34	2N3904	18kΩ	0.15µF	Fail	268Hz	312µs	2.02µA	77.29
67	cfg68	2N5179	24kΩ	0.1µF	Fail	408Hz	204µs	6.09µA	64.61
68	cfg66	2N5179	27kΩ	0.082µF	Fail	323Hz	180µs	6.06µA	64.46
69	cfg67	2N5179	27kΩ	0.1µF	Fail	265Hz	204µs	6.05µA	63.30
70	cfg60	2N5179	24kΩ	0.12µF	Fail	405Hz	252µs	6.09µA	63.15
71	cfg69	2N5179	20kΩ	0.12µF	Fail	397Hz	252µs	6.13µA	62.85
72	cfg65	2N5179	22kΩ	0.12µF	Fail	271Hz	240µs	6.11µA	62.04
73	cfg70	2N5179	18kΩ	0.15µF	Fail	265Hz	312µs	6.17µA	59.13
74	cfg58	MJE340	27kΩ	0.082µF	Fail	534Hz	168µs	16.1µA	22.54
75	cfg64	MJE340	27kΩ	0.1µF	Fail	367Hz	192µs	16.1µA	18.14
76	cfg57	MJE340	24kΩ	0.1µF	Fail	308Hz	204µs	16.1µA	16.51
77	cfg62	MJE340	22kΩ	0.12µF	Fail	430Hz	228µs	16.1µA	16.00
78	cfg63	MJE340	20kΩ	0.12µF	Fail	308Hz	228µs	16.1µA	14.19
79	cfg59	MJE340	24kΩ	0.12µF	Fail	256Hz	240µs	16.1µA	12.69
80	cfg61	MJE340	18kΩ	0.15µF	Fail	347Hz	288µs	16.1µA	8.75

REFERENCES

[1] 20-Sim. 20-Sim. http://www.20sim.com. last accessed: 04/04/2016.

[2] DEUTZ AG. http://www.deutz.com. last accessed: 04/04/2016.

[3] Environmental Protection Agency. US06 or Supplemental Federal Test Procedure. https://www.epa.gov/emission-standards-reference-guide/epa-us06-or-supplemental-federal-test-procedure-sftp. last accessed: 04/04/2016.

[4] Johan Åkesson, K-E Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and optimization with optimica and jmodelica. org?languages and tools for solving large-scale dynamic optimization problems. *Computers & Chemical Engineering*, 34(11):1737–1749, 2010.

[5] ARINC. ARINC. http://www.aeec-amc-fsemc.com/standards/index.html. last accessed: 04/04/2016.

[6] Modelica Association. Modelica Standard Library - Version 3.2.1 (Build 4). http://modelica.github.io/Modelica/, September 2015. last accessed: 04/07/2016.

[7] Modelica Association et al. The Modelica Language Specification Version 3.3 Revision 1. https://modelica.org/documents/ModelicaSpec33Revision1.pdf, July 2014. last accessed: 04/04/2016.

[8] Atlassian. JIRA. https://www.atlassian.com/software/jira/. last accessed: 04/04/2016.

[9] GbR AUTOSAR. Autosar–technical overview v2. 0.1, 2006.

[10] Brian Bailey, Grant Martin, and Thomas Anderson. *Taxonomies for the Development and Verification of digital systems.* Springer Science & Business Media, 2005.

[11] Santiago Balestrini-Robinson, Dane F Freeman, and Daniel C Browne. An object-oriented and executable sysml framework for rapid model development. *Procedia Computer Science*, 44:423–432, 2015.

[12] Ted Bapty, Justin Knight, Zsolt Lattmann, Sandeep Neema, and Jason Scott. Software quality assurance for the meta toolchain. Technical Report ISIS-15-111, Institute for Software Integrated Systems, Nashville, 01/2015 2015.

[13] Ted Bapty, Sandeep Neema, and Jason Scott. Overview of the meta toolchain in the adaptive vehicle make progam. (ISIS-15-103), 2015.

[14] Ted Bapty, Sandeep Neema, and Jason Scott. Transitioning the meta toolchain. (ISIS-15-114), 2015.

[15] Ted Bapty, Sandeep Neema, Jason Scott, and Scott Eisele. Case studies and use cases in the meta toolchain. (ISIS-15-113), 2015.

[16] Bharadwaj Bharadwaj and C Heitmeyer. Developing high assurance avionics systems with the scr requirements method. In *Digital Avionics Systems Conference, 2000. Proceedings. DASC. The 19th*, volume 1, pages 1D1–1. IEEE, 2000.

[17] Blueprintsys. Blueprint. `http://www.blueprintsys.com`. last accessed: 04/04/2016.

[18] W Borutzky, A Orsoni, and R Zobel. Bond graph modelling and simulation of mechatronic systems an introduction into the methodology. In *20th European conference on modeling and simulation*, 2006.

[19] Wolfgang Borutzky. *Bond graph methodology: development and analysis of multidisciplinary dynamic system models.* Springer Science & Business Media, 2009.

[20] Jan F Broenink. Introduction to physical systems modelling with bond graphs. *SiE Whitebook on Simulation Methodologies*, pages 1–31, 1999.

[21] Daniel Browne, Robert Kempf, Aaron Hansen, Michael O'Neal, and William Yates. Enabling systems modeling language authoring in a collaborative web-based decision support tool. *Procedia Computer Science*, 16:373–382, 2013.

[22] Manfred Broy. Towards a theory of architectural contracts: - schemes and patterns of assumption/promise based system specification. In Manfred Broy, Christian Leuxner, and Tony Hoare, editors, *Software and Systems Safety - Specification and Verification*, volume 30 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 33–87. IOS Press, 2011.

[23] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[24] Joseph T Buck, Soonhoi Ha, Edward A Lee, and David G Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, April 1994.

[25] Yue Cao, Yusheng Liu, Hongri Fan, and Bo Fan. Sysml-based uniform behavior modeling and automated mapping of design and simulation model for complex mechatronics. *Computer-Aided Design*, 45(3):764–776, 2013.

[26] Yue Cao, Yusheng Liu, and Christiaan JJ Paredis. System-level model integration of design and simulation for mechatronic systems based on sysml. *Mechatronics*, 21(6):1063–1075, 2011.

[27] Joshua D Carl, Zsolt Lattmann, and Gautam Biswas. Modeling and simulation semantics for building large-scale multi-domain embedded systems. In *27th European Conference on Modelling and Simulation*, Norway, 05 2013.

[28] Caterpillar. `http://www.cat.com`. last accessed: 04/04/2016.

[29] François E Cellier and Robert T McBride. Object-oriented modeling of complex physical systems using the dymola bond-graph library. *SIMULATION SERIES*, 35(2):157–162, 2003.

[30] François E Cellier and Dirk Zimmer. Wrapping multi-bond graphs: a structured approach to modeling complex multi-body dynamics. In *Proc. 20th European Conference on Modeling and Simulation*, pages 7–13, 2006.

[31] NASA Glenn Research Center. OpenMDAO. `http://openmdao.org`. last accessed: 04/04/2016.

[32] NASA Glenn Research Center. OpenMDAO Documentation v0.8.1. `http://openmdao.org/releases/0.8.1/docs/`, August 2013. last accessed: 04/08/2016.

[33] Inc. CircuitLab. `https://www.circuitlab.com/`. last accessed: 04/04/2016.

[34] Evin J Cramer, JE Dennis, Jr, Paul D Frank, Robert Michael Lewis, and Gregory R Shubin. Problem formulation for multidisciplinary optimization. *SIAM Journal on Optimization*, 4(4):754–776, 1994.

[35] CyDesign. CyDesign Studio. `http://cydesign.com`. last accessed: 04/04/2016.

[36] Siyuan Dai, Zsolt Lattmann, and Xenofon Koutsoukos. Compositional control design of cyber-physical systems. In Danda B Rawat, Joel Rodrigues, and Ivan Stojmenovic, editors, *Cyber Physical Systems: From Theory to Practice*. CRC Press, 2015.

[37] Pierre David, Vincent Idasiak, and Frederic Kratz. Reliability study of complex physical systems using sysml. *Reliability Engineering & System Safety*, 95(4):431–450, 2010.

[38] Luca De Alfaro and Thomas A Henzinger. Interface theories for component-based design. In *Embedded Software*, pages 148–165. Springer, 2001.

[39] Johan de Kleer. Injecting model-based diagnosis thinking into the design process.

[40] Olivier L de Weck. Feasibility of a 5x speedup in system development due to meta design. In *ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 1105–1110. American Society of Mechanical Engineers, 2012.

[41] IABG Development Standard for IT-Systems of the Federal Republic of Germany. The V-Model. `http://v-modell.iabg.de/v-modell-xt-html-english/index.html`, 2006. last accessed: 04/04/2016.

[42] Johan Eker, Jörn W Janneck, Edward Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, Yuhong Xiong, et al. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[43] Golnaz Elahi and Eric Yu. Comparing alternatives for analyzing requirements trade-offs–in the absence of numerical data. *Information and Software Technology*, 54(6):517–530, 2012.

[44] Paul Eremenko. Philosophical underpinnings of adaptive vehicle make. Technical report, DARPA-BAA-12-15. Appendix 1, 2011.

[45] Philippus J Feenstra, Pieter J Mosterman, Gautam Biswas, and Peter C Breedveld. Bond graph modeling procedures for fault detection and isolation of complex flow processes. *Simulation Series*, 33(1):77–84, 2001.

[46] Peter H Feiler, David P Gluch, and John J Hudak. The architecture analysis & design language (aadl): An introduction. Technical report, DTIC Document, 2006.

[47] Peter H Feiler, John B Goodenough, Arie Gurfinkel, Charles B Weinstock, and Lutz Wrage. Reliability validation and improvement framework. Technical report, DTIC Document, 2012.

[48] Daniel A Finke, Mark T Traband, Christopher B Ligetti, and David M Hadka. Component, context and manufacturing model library (c2m2l). Technical report, DTIC Document, 2013.

[49] Kevin Forsberg and Harold Mooz Co-Principals. 4 system engineering for faster, cheaper, better. In *INCOSE International Symposium*, volume 9, pages 924–932. Wiley Online Library, 1999.

[50] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[51] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[52] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[53] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. John Wiley & Sons, 2010.

[54] Justin Gray, Kenneth T Moore, and Bret A Naylor. Openmdao: An open source framework for multidisciplinary analysis and optimization. In *13th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Fort Worth, TX, AIAA, AIAA-2010-9101*, pages 5–7, 2010.

[55] Sol Greenspan, John Mylopoulos, and Alex Borgida. On formal requirements modeling languages: Rml revisited. In *Proceedings of the 16th international conference on Software engineering*, pages 135–147. IEEE Computer Society Press, 1994.

[56] Christopher Heath and Justin Gray. Openmdao: Framework for flexible multidisciplinary design, analysis and optimization methods. In *Proceedings of the 53rd AIAA Structures, Structural Dynamics and Materials Conference, Honolulu, HI*, 2012.

[57] Constance Heitmeyer, James Kirby, Bruce Labaw, and Ramesh Bharadwaj. Scr: A toolset for specifying and analyzing software requirements. In *Computer Aided Verification*, pages 526–531. Springer, 1998.

[58] Constance L Heitmeyer, Ralph D Jeffords, and Bruce G Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):231–261, 1996.

[59] Kathryn L Heninger. Specifying software requirements for complex systems: New techniques and their application. *Software Engineering, IEEE Transactions on*, (1):2–13, 1980.

[60] Tomonori Honda, Eric Saund, Ion Matei, William Janssen, Bhaskar Saha, Daniel G. Bobrow, Johan de Kleer, Tolga Kurtoglu, and Zsolt Lattmann. A simulation and modeling based reliability requirement assessment methodology. In *ASME 2014 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Buffalo, NY, USA, 08 2014.

[61] Christopher Hoyle, Irem Y Tumer, Tolga Kurtoglu, and Wei Chen. Multi-stage uncertainty quantification for verifying the correctness of complex system designs. In *ASME 2011 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 1169–1178. American Society of Mechanical Engineers, 2011.

[62] Ching-Lai Hwang, Young-Jou Lai, and Ting-Yun Liu. A new approach for multiple objective decision making. *Computers & operations research*, 20(8):889–899, 1993.

[63] Ching-Lai Hwang and Kwangsun Yoon. *Multiple attribute decision making: methods and applications a state-of-the-art survey*, volume 186. Springer Science & Business Media, 2012.

[64] IBM. IBM Rational DOORS. `http://www-03.ibm.com/software/products/en/ratidoor`. last accessed: 04/04/2016.

[65] IBM. IBM Rational DOORS Next Generation. `http://www-03.ibm.com/software/products/en/ratidoorng`. last accessed: 04/04/2016.

[66] Allison Transmission Inc. `http://www.allisontransmission.com/transmissions/vocational-applications/defense`. last accessed: 04/04/2016.

[67] Cummins Inc. `http://cumminsengines.com/defense`. last accessed: 04/04/2016.

[68] Vanderbilt University Institute for Software Integrated Systems. Generic Modeling Environment. `http://www.isis.vanderbilt.edu/projects/gme`. last accessed: 04/04/2016.

[69] Vanderbilt University Institute for Software Integrated Systems. OpenMETA tools. `https://vehicleforge.vf.isis.vanderbilt.edu/p/metaresources/home/`. last accessed: 04/04/2016.

[70] ITI. SimulationX. `https://www.simulationx.com`. last accessed: 04/04/2016.

[71] Ethan K Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. Components, platforms and possibilities: towards generic automation for mda. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 39–48. ACM, 2010.

[72] Ethan K Jackson, Tihamer Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In *Model Driven Engineering Languages and Systems*, pages 653–667. Springer, 2011.

[73] Laszlo Juracz, Zsolt Lattmann, Tihamer Levendovszky, Graham Hemingway, Will Gaggioli, Tanner Netterville, Gabor Pap, Kevin Smyth, and Larry Howard. Vehicleforge: A cloud-based infrastructure for collaborative model-based design. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud computing co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, volume Vol-1118, 01 2014.

[74] Dean C Karnopp, Donald L Margolis, and Ronald C Rosenberg. *System dynamics: modeling, simulation, and control of mechatronic systems*. John Wiley & Sons, 2012.

[75] Kurt Keutzer, Jan M Rabaey, A Sangiovanni-Vincentelli, et al. System-level design: orthogonalization of concerns and platform-based design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(12):1523–1543, 2000.

[76] Ekkart Kindler and Robert Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. *University of Paderborn*, (tr-ri-07-284), 2007.

[77] Matthew Evans Klenk, Johan de Kleer, Daniel G Bobrow, and Bill Janssen. Qualitative reasoning with modelica models. In *AAAI*, pages 1084–1090, 2014.

[78] Ilan Kroo, Steve Altus, Robert Braun, Peter Gage, and Ian Sobieski. Multidisciplinary optimization methods for aircraft preliminary design. *AIAA paper*, 4325:1994, 1994.

[79] Sandia National Laboratories. Dakota 6.2. `https://dakota.sandia.gov`, 2015. last accessed: 04/04/2016.

[80] Zsolt Lattmann. A multi-domain functional dependency modeling tool based on extended hybrid bond graphs. M.sc, Vanderbilt University, 04 2010.

[81] Zsolt Lattmann, James Klingler, Patrik Meijer, Ted Bapty, Sandeep Neema, and Jason Scott. Integration platform technology components in the meta toolchain. Technical Report ISIS-15-110, Institute for Software Integrated Systems, Nashville, 01/2015 2015.

[82] Zsolt Lattmann, James Klingler, Patrik Meijer, Ted Bapty, Sandeep Neema, and Jason Scott. Meta design space exploration using dynamics. Technical Report ISIS-15-106, Institute for Software Integrated Systems, Nashville, 01/2015 2015.

[83] Zsolt Lattmann, James Klingler, Patrik Meijer, Jason Scott, Sandeep Neema, Ted Bapty, and Gábor Karsai. Towards an Analysis-Driven Rapid Design Process for Cyber-Physical Systems. *26th IEEE International Symposium on Rapid System Prototyping (RSP)*, 10 2015.

[84] Zsolt Lattmann, Adam Nagel, Tihamer Levendovszky, Ted Bapty, Sandeep Neema, and Gabor Karsai. Component-based modeling of dynamic systems using heterogeneous composition. In *6th International Workshop on Multi-Paradigm Modeling*, Innsbruck, Austria, 10 2012.

[85] Zsolt Lattmann, Adam Nagel, Jason Scott, Kevin Smyth, Johanna Ceisel, Chris vanBuskirk, Joseph Porter, Sandeep Neema, Ted Bapty, Dimitri Mavris, and Janos Sztipanovits. Towards automated evaluation of vehicle dynamics in system-level designs. In *Proc. ASME International Design Engineering Technical Conf. & Computers and Information in Engineering Conf. (IDETC/CIE 2012)*, Chicago, IL, USA, 08 2012.

[86] Zsolt Lattmann, Adrian Pop, Johan de Kleer, Peter Fritzson, Bill Janssen, Sandeep Neema, Ted Bapty, Xenofon Koutsoukos, Matthew Klenk, Daniel Bobrow, Bhaskar Saha, and Tolga Kurtoglu. Verification and design exploration through meta tool integration with openmodelica. In *Proceedings of the 10th International Modelica Conference*, pages 353–362, Lund University, olvegatan 20A, SE-223 62 LUND, SWEDEN, 03 2014. Modelica Association and Linoping University Electronic Press.

[87] Edward Lee et al. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE, 2008.

[88] Edward A Lee. Cyber-physical systems-are computing foundations adequate. In *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, volume 2. Citeseer, 2006.

[89] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach.* http://leeseshia.org, Second edition, 2015.

[90] Sparx Systems Pty Ltd. Enterprise Architect. `http://www.sparxsystems.com/products/ea/index.html`. last accessed: 04/04/2016.

[91] Mark W Maier. *The art of systems architecting.* CRC press, 2009.

[92] Brett Malone and Michael Papay. Modelcenter: an integration environment for simulation based design. In *Simulation Interoperability Workshop*, 1999.

[93] MapleSoft. MapleSim. `http://www.maplesoft.com/products/maplesim/whychoose.aspx`. last accessed: 04/04/2016.

[94] MathWorks. SimScape. `http://www.mathworks.com/help/physmod/simscape/`. last accessed: 04/04/2016.

[95] MathWorks. SimScape Language Guide. `http://www.mathworks.com/help/pdf_doc/physmod/simscape/simscape_lang.pdf`. last accessed: 04/04/2016.

[96] MathWorks. Simulink. `http://www.mathworks.com/products/simulink/`. last accessed: 04/04/2016.

[97] Robert T McBride and François E Cellier. System efficiency measurement through bond graph modeling. In *Proc. ICBGM?05 Conference*, 2005.

[98] Stephen J Mellor, Marc Balcer, and Ivar Foreword By-Jacoboson. *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[99] MetaMorph Software, Inc. MetaMorphosys. `http://www.metamorphsoftware.com/metamorph-tools/`. last accessed: 04/27/2016.

[100] Modelon. JModelica.org. `http://www.jmodelica.org`. last accessed: 04/04/2016.

[101] Pieter J Mosterman. *Hybrid Dynamic Systems: A hybrid bond graph modeling paradigm and its application in diagnosis*. PhD thesis, Vanderbilt University, 1997.

[102] Amalendu Mukherjee and Arun Kumar Samantaray. *Bond graph in modeling, simulation and fault identification*. IK International Pvt Ltd, 2006.

[103] David J Musliner and Eric Engstrom. Prismatic: Unified hierarchical probabilistic verification tool. Technical report, DTIC Document, 2011.

[104] Adam Nagel. AVM Class Diagrams. `https://github.com/metamorph-inc/meta-core/tree/master/meta/DesignDataPackage/doc/ClassDiagrams`, 2014. last accessed: 04/04/2016.

[105] Adam Nagel, Sandeep Neema, Mike Myers, Robert Owens, Zsolt Lattmann, and Dan Finke. AVM Component Specification version 2.5. `https://github.com/metamorph-inc/meta-core/blob/master/meta/DesignDataPackage/doc/AVM_Component_Spec.pdf`, 2014. last accessed: 04/04/2016.

[106] Himanshu Neema, Jesse Gohl, Zsolt Lattmann, Janos Sztipanovits, Gabor Karsai, Sandeep Neema, Ted Bapty, John Batteh, Hubertus Tummescheit, and Chandrasekar Sureshkumar. Model-based integration platform for fmi co-simulation and heterogeneous simulations of cyber-physical systems. In *Proceedings of the 10th International Modelica Conference*, pages 235–245, Lund University, olvegatan 20A, SE-223 62 LUND, SWEDEN, 03 2014. Modelica Association and Linoping University Electronic Press.

[107] Himanshu Neema, Zsolt Lattmann, Patrik Meijer, James Klingler, Sandeep Neema, Ted Bapty, Janos Sztipanovits, and Gabor Karsai. Design space exploration and manipulation for cyber physical systems. In *IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems (IDEAL' 2014)*, Berlin, Germany, 04 2014. Springer, Springer.

[108] Himanshu Neema, Sandeep Neema, and Ted Bapty. Architecture exploration in the meta toolchain. (ISIS-15-105), 2015.

[109] Sandeep Neema, Ted Bapty, and Daniel Balasubramanian. Software design and implementation in the meta toolchain. (ISIS-15-107), 2015.

[110] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-based design-space exploration and model synthesis. In *EMSOFT 2003, LNCS 2855*, Philadelphia, PA, October 2003.

[111] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and Ken Butts. Constraint-based design-space exploration and model synthesis. In *Embedded Software*, pages 290–305. Springer, 2003.

[112] Pierluigi Nuzzo. *Compositional Design of Cyber-Physical Systems Using Contracts*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2015.

[113] OMG. Business Process Model And Notation. `http://www.omg.org/spec/BPMN/2.0/`, 2011. last accessed: 04/04/2016.

[114] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.4. `http://www.omg.org/spec/SysML/1.4/`, 2015.

[115] OMG. Unified Modeling Language (UML), Version 2.5. `http://www.omg.org/spec/UML/2.5/`, 2015.

[116] Tim O'reilly. What is web 2.0: Design patterns and business models for the next generation of software. *Communications & strategies*, (1):17, 2007.

[117] Open Source Modelica Consortium (OSMC). OpenModelica. `https://www.openmodelica.org`. last accessed: 04/04/2016.

[118] M. Otter, H. Elmqvist, and S. E. Mattsson. Multidomain Modeling with Modelica. In Paul A. Fishwick, editor, *Handbook of Dynamic System Modelling*, chapter 36, pages 36.1 – 36.27. Chapman & Hall/CRC, 2007.

[119] Hewlett Packard. HP Quality Center. `http://www8.hp.com/us/en/software-solutions/quality-center-quality-management/`. last accessed: 04/04/2016.

[120] C Paredis. Model-based systems engineering: A roadmap for academic research. *Frontiers in Model-Based Systems Engineering, Atlanta, GA*, 2011.

[121] Roberto Passerone, Luca De Alfaro, Thomas A Henzinger, and Alberto L Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 132–139. ACM, 2002.

[122] Fabio Paterno. *Model-based design and evaluation of interactive applications*. Springer Science & Business Media, 2012.

[123] R Peak, R Burkhart, S Friedenthal, M Wilson, M Bajaj, and I Kim. Simulation-based design using sysml part 1: A parametrics primer. incose intl. In *Symposium, San Diego*, 2007.

[124] Russell S Peak, Roger M Burkhart, Sanford A Friedenthal, Miyako W Wilson, Manas Bajaj, and Injoong Kim. 9.3. 3 simulation-based design using sysml part 2: Celebrating diversity by example. In *INCOSE International Symposium*, volume 17, pages 1536–1557. Wiley Online Library, 2007.

[125] Linda R Petzold et al. A description of dassl: A differential/algebraic system solver. *Scientific computing*, 1, 1982.

[126] Klaus Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010.

[127] Adrian Pop, Vasile Băluţă, and Peter Fritzson. Eclipse support for design and requirements engineering based on modelicaml. *SIMS 2007*, page 93, 2007.

[128] Joseph Porter, Zsolt Lattmann, Graham Hemingway, Nagabhushan Mahadevan, Sandeep Neema, Harmon Nine, Nicholas Kottenstette, Peter Volgyesi, Gabor Karsai, and Janos Sztipanovits. The esmol modeling language and tools for synthesizing and simulating real-time embedded systems. In *15th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, CA, 04 2009.

[129] PTC. PTC Integrity. `http://www.ptc.com/application-lifecycle-management/integrity`. last accessed: 04/04/2016.

[130] Sharif Rahman and Heqin Xu. A univariate dimension-reduction method for multi-dimensional integration in stochastic mechanics. *Probabilistic Engineering Mechanics*, 19(4):393–408, 2004.

[131] 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*, pages 731–736. ACM, 2010.

[132] Bhaskar Saha, Tomonori Honda, Ion Matei, Eric Saund, Johan de Kleer, Tolga Kurtoglu, and Zsolt Lattmann. A model-based approach for an optimal maintenance strategy. In *2nd European Conference of the PHM Society - PHME'14*, Nantes, France, 07 2014.

[133] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems*. *European journal of control*, 18(3):217–238, 2012.

[134] Ulf Sellgren. Simulation driven design-a functional view of the design process. *Licentiate thesis. Royal institute of technology, Sweden*, 1995.

[135] Ulf Sellgren. Simulation-driven design: motives, means, and opportunities. 1999.

[136] Gabor Simko. *Formal Semantic Specification of Domain-Specific Modeling Languages for Cyber-Physical Systems.* PhD thesis, Vanderbilt University, 2014.

[137] Timothy W Simpson, Timothy M Mauery, John J Korte, and Farrokh Mistree. Kriging models for global approximation in simulation-based multidisciplinary design optimization. *AIAA journal*, 39(12):2233–2241, 2001.

[138] Timothy W Simpson, Vasilli Toropov, Vladimir Balabanov, and Felipe AC Viana. Design and analysis of computer experiments in multidisciplinary design optimization: a review of how far we have come or not. In *12th AIAA/ISSMO multidisciplinary analysis and optimization conference*, volume 5, pages 10–12, 2008.

[139] Rajarishi Sinha, Christiaan JJ Paredis, Vei-Chung Liang, and Pradeep K Khosla. Modeling and simulation methods for design of engineering systems. *Journal of Computing and Information Science in Engineering*, 1(1):84–91, 2001.

[140] Ian Sommerville and Pete Sawyer. *Requirements engineering: a good practice guide*. John Wiley & Sons, Inc., 1997.

[141] Sara C Spangelo, David Kaslow, Chris Delp, Bjorn Cole, Louise Anderson, Elyse Fosse, Brett Sam Gilbert, Leo Hartman, Theodore Kahn, and James Cutler. Applying model based systems engineering (mbse) to a standard cubesat. In *Aerospace Conference, 2012 IEEE*, pages 1–20. IEEE, 2012.

[142] Rolls-Royce Power Systems. `http://www.mtu-online.com/mtu/mtu/`. last accessed: 04/04/2016.

[143] Dassault Systèmes. Catia. `http://www.3ds.com/products-services/catia`. last accessed: 04/04/2016.

[144] Dassault Systèmes. Dymola. `http://www.3ds.com/products-services/catia/products/dymola`. last accessed: 04/04/2016.

[145] Dassault Systèmes. iSight 5.9-2. `http://www.3ds.com/products-services/simulia/products/isight-simulia-execution-engine/`. last accessed: 04/04/2016.

[146] J. Sztipanovits, T. Bapty, S. Neema, X. Koutsoukos, and E. Jackson. Design tool chain for cyber-physical systems: Lessons learned. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6, June 2015.

[147] Janos Sztipanovits. Cyber physical systems: New challenges for model-based design, April 2008. Presented at the " <a href="http://chess.eecs.berkeley.edu/conferences/08/StLouis/index.htm" >From Embedded Systems to Cyber-Physical Systems: a Review of the State-of-the-Art and Research Needs</a>" Workshop, St. Louis, MO.

[148] Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan Jackson. Openmeta: A model-and component-based design tool chain for cyber-physical systems. In *From Programs to Systems. The Systems perspective in Computing*, pages 235–248. Springer, 2014.

[149] Michael Tiller. *Introduction to physical modeling with Modelica*, volume 615. Springer Science & Business Media, 2012.

[150] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.

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

[152] Tim Weilkiens. *Systems engineering with SysML/UML: modeling, analysis, design*. Morgan Kaufmann, 2011.

[153] J.C. Willems. The behavioral approach to open and interconnected systems. *Control Systems, IEEE*, 27(6):46 –99, Dec 2007.

[154] Wolfram. SystemModeler. `http://www.wolfram.com/system-modeler`. last accessed: 04/04/2016.

[155] Thought Works. Mingle. `https://www.thoughtworks.com/mingle/`. last accessed: 04/04/2016.

[156] Ryan Wrenn, Adam Nagel, Robert Owens, Di Yao, Himanshu Neema, Feng Shi, Kevin Smyth, Joseph Porter, Ted Bapty, Sandeep Neema, et al. Towards automated exploration and assembly of vehicle design models. In *ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 1143–1152. American Society of Mechanical Engineers, 2012.

[157] Dazhong Wu, Linda L Zhang, Roger J Jiao, and Roberto F Lu. Sysml-based design chain information modeling for variety management in production reconfiguration. *Journal of Intelligent Manufacturing*, 24(3):575–596, 2013.

[158] Kwangsun Yoon. A reconciliation among discrete compromise solutions. *Journal of the Operational Research Society*, pages 277–286, 1987.

[159] Peng Zhang, Zsolt Lattmann, James Klingler, Sandeep Neema, and Ted Bapty. Visualization techniques in collaborative domain-specific modeling environment. In *SoutheastCon 2015*, Fort Lauderdale, FL, USA, 04/2015 2015. IEEE.