SYNGRESS
ADVANCED TOPICS
IN INFORMATION SECURITY

HANDBOOK OF SYSTEM SAFETY AND SECURITY

Cyber Risk and Risk Management, Cyber Security, Threat Analysis, Functional Safety, Software Systems, and Cyber Physical Systems

Edited by
Edward Griffor

HANDBOOK OF SYSTEM SAFETY AND SECURITY

Cyber Risk and Risk Management, Cyber Security, Threat Analysis, Functional Safety, Software Systems, and Cyber Physical Systems

Edited by

EDWARD GRIFFOR
National Institute of Standards and Technology (NIST),
Gaithersburg, MD, United States

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON • NEW YORK • OXFORD
PARIS • SAN DIEGO • SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Syngress is an imprint of Elsevier

SYNGRESS.

Syngress is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

ISBN: 978-0-12-803773-7

Publisher: Todd Green
Acquisition Editor: Chris Katsaropoulos
Editorial Project Manager: Anna Valutkevich
Production Project Manager: Punithavathy Govindaradjane
Cover Designer: Mark Rogers

Typeset by MPS Limited, Chennai, India

CONTENTS

ABOUT THE EDITOR

Dr. Edward Griffor is the Associate Director for Cyber-Physical Systems at the National Institute of Standards and Technology (NIST) in the US Department of Commerce. Prior to joining NIST in July of 2015, he was a Walter P. Chrysler Technical Fellow, one of the highest technical positions in the automotive industry and one that exists in multiple industry sectors, including transportation, aerospace, science, defense, energy, and medical. He served as Chairman of the Chrysler Technology Council until 2015 and continues to serve as Chairman of The MIT Alliance, a professional association of scientists, engineers, and business experts trained at the Massachusetts Institute of Technology.

He completed doctoral studies in Mathematics at MIT and was awarded Habilitation by the Mathematics and Engineering Faculty of the University of Oslo. He was named National Science Foundation/NATO Postdoctoral Fellow in Science and Engineering in 1980. He was on the faculty of Uppsala University in Uppsala, Sweden, from 1980 to 1997 and returned to the United States to lead advanced research in Electrical Engineering in the automotive industry.

He has been on the faculties of the University of Oslo in Norway, Uppsala University in Sweden, the Catholic University of Santiago in Chile as well as those of Harvard, MIT, and Tufts University in the United States. He is regarded as one of the world experts in the use of mathematical methods for the design and assurance of technologies used in developing advanced, adaptive cyber-physical systems, including those used to ensure the safety and security of autonomous systems. In addition to his work at Chrysler, he has led research in biosystem modeling and simulation. He is an Adjunct Professor at the Wayne State University School of Medicine in Detroit, MI, at the Center for Molecular Medicine and Genetics.

His work in the automotive industry provided advanced algorithms for Voice Recognition and Autonomous and Connected Vehicles. He has published three books previously, including *Handbook of Computability* by Elsevier, *Theory of Domains*, by Cambridge University Press, and *Logic's Lost Genius: The Life of Gerhard Gentzen*, by American Mathematical Society. He has published extensively in professional journals and has given invited presentations for the American Mathematical Society, Association for Symbolic Logic, North American Software Certification Consortium, Society of Automotive Engineers, the Federal Reserve Bank, and US government agencies, including NIST, DARPA, DOE, DOT, and NASA.

ABOUT THE CONTRIBUTORS

Ted Bapty is a Research Associate Professor and Senior Researcher at the Institute for Software Integrated Systems. He is interested in and leads research projects in Model-Integrated Systems as applied to: Cyber Physical System Design, Large-Scale & Distributed Real-Time Embedded Systems, C4ISR systems, Digital Signal Processing and Instrumentation Systems, and tools for Rapid System Prototyping and System Integration. Current and recent projects include DARPA AVM/META Cyber-Physical Design Tools and Model-Based tools for the Future Airborne Capabilities Environment (FACE) Standard. He holds a BSEE from the University of Pennsylvania and a PhD from Vanderbilt University, and has served as a Captain in the US Air Force. He is cofounder of Metamorph Software, a spin-off company formed to transition model-based engineering tools.

Abdella Battou is the Division Chief of the Advanced Network Technologies Division, within The Information Technology Lab at NIST. He also leads the Cloud Computing Program. Before joining NIST in 2012, he served as the Executive Director of The Mid-Atlantic Crossroads (MAX) GigaPop founded by The University of Maryland, The George Washington University, The Georgetown University, and The Virginia Polytechnic Institute. From 2000 to 2009, he was Chief Technology Officer, and Vice President of Research and Development for Lambda Optical Systems, where he was responsible for overseeing the company's system architectures, hardware design, and software development teams. Additionally, he served as senior research scientist for the Naval Research Laboratory's high-speed networking group, Center for Computational Sciences from 1992 to 2000. He holds a PhD and MSEE in Electrical Engineering from the Catholic University of America.

Monika Bialy is a PhD student in the Department of Computing and Software at McMaster University, Hamilton, ON, Canada. She received her master's degree in software engineering ("http://MASc" MASc) from McMaster University in 2014, and Honours Bachelor of Computer Science ("http://BCoSc" BCoSc) in 2012 from Laurentian University, Sudbury, ON, Canada. Monika currently holds an NSERC Alexander Graham Bell Canada Graduate Scholarship-Doctoral (CGS D). Her main research interests include model-based development, safety-critical systems, and software engineering design principles.

Hasnae Bilil was born in Rabat, Morocco, in 1986. She received the Dipl.-Ing in 2010 and the PhD degree in 2014 in electrical engineering from Mohammadia School of Engineers, Rabat, Morocco. She is now a teaching assistant at Mohammadia School of Engineers, University Mohammed V in Rabat, Morocco. Since August 2015, she has been conducting research on “Smart grid” and “Information-centric networking” as guest researcher at National Institute of Standards and Technology. Her current research interests include renewable energy sources, power system, smart grid, and power management into a power system integrating renewable energy source.

Chris Greer is Senior Executive for Cyber Physical Systems, Director of the Smart Grid and Cyber-Physical Systems Program Office, and National Coordinator for Smart Grid Interoperability at the National Institute of Standards and Technology. Prior to joining NIST, he served as Assistant Director for Information Technology R&D in the White House Office of Science and Technology Policy (OSTP) and Cybersecurity Liaison to the National Security Staff. His responsibilities there included networking and information technology, research and development, cybersecurity, and digital scientific data access. He has also served as Director of the National Coordination Office for the Federal Networking and Information Technology Research and Development (NITRD) Program. This program coordinates IT R&D investments across the Federal government, including the cyber-physical systems research portfolio.

Salim Hariri is the Director of the NSF Center for Cloud and Autonomic Computing and Professor in the Electrical and Computer Engineering Department at the University of Arizona, 2004 to present. He holds PhD from the Computer Engineering Dept. in University of Southern California, Los Angeles, CA, and MSc from Electrical Engineering in The Ohio State University, Columbus, OH. His research areas include, but not limited to, autonomic computing, self-protection of networks and computers, high-performance distributed computing, cyber security, proactive network management, cloud computing, resilient system architecture, Internet of Things (IoT).

Michael Huth is Professor of Computer Science, Director of Research, and Head of the Security Research Group in the Department of Computer Science at Imperial College London. He is a Diplom-Mathematiker (TU Darmstadt, Germany), obtained his PhD in 1991 (Tulane University of Louisiana, USA), and completed several postdoctoral studies in the United States, Germany, and the United Kingdom on programming language semantics and design, formal verification, and probabilistic modeling. His present research focuses on cybersecurity, especially modeling and reasoning about the interplay of trust, security, risk, and economics. Currently funded projects of his include work on confidence building in arms verification and work on blockchain technology for centrally governed systems such as IoT. He is a member of the ACM and active as research and product advisor in the London Cybersecurity startup scene.

Jason Jaskolka is a US Department of Homeland Security Cybersecurity Postdoctoral Scholar at Stanford University within the Center for International Security and Cooperation (CISAC). He received his PhD in Software Engineering in 2015 from McMaster University, Hamilton, ON, Canada. His research interests include cybersecurity assurance, distributed multiagent systems, and algebraic approaches to software engineering.

James M. Kaplan is a partner with McKinsey & Company in New York. He leads McKinsey's global Cybersecurity Practices and server banks, manufacturers, and health institutions on a range of technology issues. In addition to publishing on enterprise technology topics in the *McKinsey Quarterly, McKinsey on Business Technology,* the *Wall Street Journal,* and the *Financial Times,* he is also the lead author of *Beyond Cybersecurity: Protecting Your Digital Business.*

Siham Khoussi graduated from the Mohammadia School of Engineers (EMI) as an Electrical Engineer majored in Automation and Industrial Computer Science. She has worked with the Research Institute for Solar Energy and New Energies (IRESEN). She is currently working at the National Institute of Standards and Technology (NIST). Her research of interests include smart grid and renewable energies, smart cities, Named Data networks (NDN), and Network verification.

Zsolt Lattmann is currently a Staff Engineer II at the Institute for Software Integrated Systems at Vanderbilt University. He has an undergraduate degree in Electrical Engineering from Budapest University of Technology and Economics in Hungary (2009), MSc and PhD degrees from Vanderbilt University in Nashville, TN, in 2010 and 2016, respectively. He was one of the lead developers on the META project of the Adaptive Vehicle Make program sponsored by DARPA between 2010 and 2014. He had joined this project in 2010 and had been researching, developing, and implementing solutions in a metamodel-based environment using various domain models and applications. He integrated an open-source optimization tool (OpenMDAO) to the OpenMETA tool chain to provide a higher level of abstraction for end users. He is currently the Principal Investigator of the WebGME project since 2015. WebGME is an open-source Web-based collaborative metamodeling environment. Domain-specific languages and tools can be developed using WebGME to improve engineer's productivity and reduce design time and cost. His primary interest includes electrical, mechanical, multibody, fluid, and thermal domains in modeling, simulation, and parametric and discrete design space studies. He has experience with the OpenMETA tool chain and WebGME, developing new domain-specific modeling languages, and implementing model transformation tools.

Mark Lawford is a Professor in McMaster University's Department of Computing and Software and the Associate Director of the McMaster Centre for Software Certification. He is a licensed Professional Engineer in the province of Ontario and a Senior Member of the IEEE. He received his PhD in 1997 from the Systems Control Group in Electrical and Computer Engineering at the University of Toronto and then worked at Ontario Hydro as a real-time software verification consultant on the Darlington Nuclear Generating Station Shutdown Systems Redesign project, receiving the Ontario Hydro New Technology Award for Automation of Systematic Design Verification of Safety Critical Software in 1999. He joined McMaster University's Department of Computing and Software in 1998 where he helped to develop the Software Engineering programs and Mechatronics Engineering programs. He served as the Section Chair for Computer Systems on the Computer Science Evaluation Group for the 2010 NSERC Discovery Grant Competition. From 2006 to 2007, he was a Senior Researcher in the Software Quality Research Lab at the University of Limerick, and in August 2010, he was a visiting researcher at the Center for Devices and Radiological Health, Office of Science and Engineering Laboratories of the US FDA. In 2014 he was a corecipient of the Chrysler Innovation Award for his work with Dr. Ali Emadi on the Automotive Partnership Canada (APC) project entitled "Next Generation Affordable Electrified Powertrains with Superior Energy Efficiency and Performance-Leadership in Automotive Powertrain (LEAP)." His research interests include software certification, application of formal methods to safety critical real-time systems, supervisory control of discrete event systems, and cyber physical systems.

Charif Mahmoudi received the MSc and PhD degrees in computer engineering from the University of Paris-EST (France) in 2009 and 2014, respectively. Since then, he has been a PostDoc at the National Institute of Standards and Technology. He participated as consultant then software architect to several successful telecommunication projects within France Telecom and Bouygues Telecom. His areas of research are on distributed systems, cloud-computing, mobile computing, and IoT.

Riccardo Masucci is a public policy professional. He currently works as Senior Manager at Intel Corporation and leads the activities related to data protection and cybersecurity policies in Europe, Middle East, and Africa. He previously served as policy advisor to Members of the Justice and Home Affairs Committee in the European Parliament. He studied in Italy and Austria and he holds a Master's Degree in International Relations.

Andreas Mattas is a member of the teaching stuff of the School of Economic Sciences of the Aristotle University of Thessaloniki. He holds a Diploma in Applied Mathematics of Aristotle University of Thessaloniki, Greece, and a Doctor's degree (PhD) in Information Security from the Aristotle University of Thessaloniki, Greece. His research interests include information security, information modeling and optimization.

Joseph D. Miller has served as the chairman of the United States Technical Advisory Group since 2005 which developed ISO 26262: Road Vehicles – Functional Safety. This was recognized by the SAE Technical Standards Board Outstanding Contribution Award. He provided the Technical Keynote at the Safety Critical Systems sessions of the 2011 SAE World Congress, teaches an SAE Webinar introduction to ISO 26262, and serves on the boards for the VDA safety conference in Berlin and the CTI safety conference in the United States. He is the Chief Engineer of Systems Safety at TRW Automotive responsible for the systems safety process. Prior to this, he has managed systems engineering, manufacturing planning, and program control for electric steering. He has also engineered communication, avionics, infrared, and radar systems, as well as and thick and thin film components. He has 20 US patents, a Master of Engineering (EE), and a Master of Business Administration.

Sandeep Neema is a Research Associate Professor of Electrical Engineering and Computer Science at Vanderbilt University, and a Senior Research Scientist at Institute for Software Integrated Systems. His research interests include Cyber Physical Systems, Model-based Systems Design and Integration, Mobile Computing, and Distributed Computing. He received his PhD from Vanderbilt University in 2001.

Vera Pantelic received the BEng in Electrical Engineering from the University of Belgrade, Belgrade, Serbia, in 2001, and MASc and PhD in Software Engineering from McMaster University, Hamilton, ON, Canada, in 2005 and 2011, respectively. She is working as a Principal Research Engineer with the McMaster Centre for Software Certification, and McMaster Institute for Automotive Research and Technology (MacAUTO), McMaster University. Her research interests include development and certification of safety-critical software systems, model-based design, and supervisory control of discrete event systems.

Lucian Patcas is a Postdoctoral Fellow in the Department of Computing and Software at McMaster University in Hamilton, ON, Canada, and also a Principal Research Engineer with the McMaster Centre for Software Certification (McSCert) and McMaster Institute for Automotive Research and Technology (MacAUTO). His main research interests lie in the area of formal methods for real-time and safety-critical software. Currently, he is involved in several research projects related to the safety of automotive software, simulation of CAN networks, and model-based development of automotive software. He received his PhD in Software Engineering from McMaster University in 2014, master's in Computer Science from University College Dublin, Ireland in 2007, and bachelor's in Software Engineering from Politehnica University of Timisoara, Romania in 2004.

Andrea Piovesan was born in Italy and received his Master of Science degree in Engineering Physics from the University of

Turin, Torino, Italy. He has started his professional career at Fiat Research Centre where he gained over 10 years' experience in safety and reliability of embedded electronic systems, for automotive and aeronautic industries. Always looking for new challenges in applying new processes and innovative technologies, Andrea is an R&D specialist focused on the development of complex, safety-critical systems. After a long experience spent on by-wire systems and innovative powertrain systems, he was assigned to the ISO Working Group 16 as a technical expert for the development of the automotive functional safety standard ISO 26262. Andrea is Functional Safety Expert at Metatronix S.r.l, a company of the Metatron Group, worldwide leader in research and development of Engine Control Systems dedicated to CNG, LNG, and LPG alternative fuels.

Alexander Schaap received his bachelor's degree in Computer Science in the Netherlands in 2013. After returning to Canada, he continued his studies, doing a master's degree in software engineering at McMaster University. He is currently a part of Leadership in Automotive Powertrain (LEAP) project. His research interests include not only the application of generative programming techniques and functional programming languages but also proper software engineering as a whole.

Dr. Anuja Sonalker, PhD, is founder of STEER auto cyber, where she leads development of cyber security for advanced and future vehicles. Prior to STEER she was Vice President of Engineering & Operations, North America, for TowerSec where she led engineering, operations, and market facing R&D for the North American market. She established the global engineering services division and led several new business contracts. She is an expert in cyber security for embedded and distributed networked systems. She brings together a broad set of technical skills, demonstrated leadership, and experience from working with government, academia, and industry leaders. She has led various efforts in the past 16 + years in automotive cyber security, intrusion detection, Internet infrastructure security, wireless systems security, sensor networks, security protocol design, and cryptography. She is currently the Vice Chair of the SAE Committee on Automotive Security Guidelines and Risk Development under Electrical Systems. Prior to TowerSec, she led innovation in automotive cyber security at Battelle. At Battelle she was co-inventor of the world's first and only Sigma Six accurate Intrusion Detection System for cars. She holds two patents in the area of automotive cyber security. She also executed field trials on decoupled projects with several automakers paving the way for carmakers to accept IDS as a necessity and issue requirements. She maintained industry outreach and was invited speaker to several technical and nontechnical venues across the world on automotive cyber security issues. She served as an advisory member of the Battelle Senior Technical Council. Prior to Battelle she worked as a PI/Branch Chief at Sparta, and was a Research Staff Member of security at IBM TJ Watson, and Fujitsu Labs. She had worked in various security domains during the time from Internet Infrastructure to wireless handhelds, and enterprise security. During this time, she was also a contributing author to several standardization activities including IEEE 802.11S, ANSI T11 cyber security, and IETF Secure Inter Domain Routing (SIDR). She completed her doctoral studies from the University of Maryland, College Park, in Electrical Engineering with her thesis in Wireless Distributed Systems Security. Her thesis was on securing collaborative services in wireless sensor networks in highly adversarial scenarios. In her spare time, she mentors high school kids toward STEM disciplines and women through the Scholarships for Women Studying Information Systems (SWSIS).

Dr. Janos Sztipanovits is currently the E. Bronson Ingram Distinguished Professor of Engineering at Vanderbilt University and founding director of the Vanderbilt Institute for Software Integrated Systems. Between 1999 and 2002, he worked as program manager and acting deputy director of DARPA Information Technology Office. He leads the CPS Virtual Organization and he is co-chair of the CPS Reference Architecture and Definition public working group established by NIST in 2014. In 2014/15 he served as academic member of the Steering Committee of the Industrial Internet Consortium. He was elected Fellow of the IEEE in 2000 and external member of the Hungarian Academy of Sciences in 2010.

Dr. Cihan Tunc is a Research Assistant Professor in the Electrical and Computer Engineering Department at the University of Arizona and associated with the Autonomic Computing Lab (ACL) in the University of Arizona. He holds PhD from the Electrical and Computer Engineering Department of the University of Arizona. His research areas include autonomic power, performance, and security management for the cloud computing systems, IoT, and cyber security.

Claire Vishik is Trust & Security Director at Intel Corporation. Her work focuses on hardware security, Trusted Computing, privacy enhancing technologies, and some aspects of encryption and related policy issues. She is a member of the Permanent Stakeholders Group of the European Network and Information Security Agency (ENISA). She holds leadership positions in standards development and is on the Board of Directors of the Trusted Computing Group (TCG) and a Council Member of the Information Security Forum. She is an active member of research organizations and initiatives; she is a Board member for Trust in Digital Life (TDL) and member of the Cybersecurity Steering Group for the UK Royal Society. She serves on advisory and review boards of a number of research initiatives in security and privacy in Europe and the United States. Prior to joining Intel, she worked at Schlumberger Laboratory for Computer Science and AT&T Laboratories. She is the author of a large number of peer-reviewed papers, as well as an inventor on 30 + pending and granted US patents. She received her PhD from the University of Texas at Austin.

Dr. Alan Wassyng is the Director of the McMaster Centre for Software Certification (McSCert). He has been working on safety-critical software-intensive systems for more than 25 years, and is licensed as a Professional Engineer in Ontario. After spending 14 years as an academic, he consulted independently on critical software development for more than 15 years. He helped Ontario Hydro (OH) develop methods for safety-critical systems, and was a key member of the team that designed the methodology and built the software for the shutdown systems for the Darlington Nuclear Station. In 1995 he was awarded an OH New Technology Award for "Development of Safety-Critical Software Engineering Technology." In 2002 he returned to academia. He publishes on software certification, and the development of safe and dependable software-intensive systems. He is a cofounder of the Software Certification Consortium (SCC), and has served as Chair of the SCC Steering Committee since its inception in 2007. He has consulted for the US Nuclear Regulatory Commission, and in July 2011, he was a visiting researcher in the Center for Devices and Radiological Health at the US Federal Drug Administration. In 2012 he was invited to give a keynote talk at Formal Methods (the premier conference in the field), and a keynote at FormaliSE 2013. In 2006 he was awarded the McMaster Students Union Award for Teaching Excellence in the Faculty of Engineering. He has served as a PI or co-PI on a number of funded projects at McMaster University.

This page intentionally left blank

INTRODUCTION

C. Greer
National Institute of Standards and Technology, Gaithersburg, MD, United States

With expectations for between 50 and 200 billion connected devices worldwide by 2020, the global Internet of Things market is predicted to expand at a compound annual growth rate of over 31%, exceeding $9T by the 2020 milestone.[1]

Internet of Things (IoT) concepts are expected to drive progress across nearly all sectors of the global economy. GE estimates of the Industrial Internet could add $10T to $15T to global GDP over the next 20 years. Gartner predicts that there will be 250 million connected vehicles on the road by 2020. Navigant predicts that the worldwide installed base of smart meters will grow from over 300 million today to more than 1 billion by 2022. IDC predicts that the wearable connected fitness device market will grow from 45 million units in 2015 to 126 million in 2019.

The impact of networking and information technology (NIT) is stunning. Virtually every human endeavor is affected as advances in NIT enable or improve domains such as scientific discovery, human health, education, the environment, national security, transportation, manufacturing, energy, governance, and entertainment.[2]

Realizing the full benefits of these emerging IoT concepts will require advances in science and engineering to meet the grand challenges posed by emerging IoT applications in terms of scale, connectivity, complexity, and interdependence. The numbers above speak to the issue of scale. The largest growth in connectivity is expected for devices not traditionally network-connected—devices like home thermostats, street lights,

[1]See, for example, http://www.intel.com/content/dam/www/public/us/en/images/iot/guide-to-iot-infographic.png; http://www.technavio.com/pressrelease/the-global-internet-of-things-market-is-expected-to-grow-at-a-cagr-of-3172-percent.
[2]President's Council of Advisors on Science and Technology, Designing a Digital Future: Federally Funded Research and Development in Networking and Information Technology, January 2013.

and automobiles—creating new markets for systems-of-systems designs. This connectivity is often multinodal—a connected vehicle may interact not just with the driver, but with other vehicles, road infrastructure, transportation management systems, public safety systems, and more—creating increased levels of complexity and new interdependencies.

These new connections and interdependencies create new safety and security concerns. Connectivity means that physical incidents in IoT systems may arise not only from physical means but from cybersources as well, increasing the attack vectors for important infrastructures with significant economic and life safety implications. And new interdependencies mean that a failure or an attack may not be limited to a single technology or sector.

> *Removing the cyber-physical barriers in an urban environment [smart city] presents a host of opportunities for increased efficiencies and greater convenience, but the greater connectivity also expands the potential attack surface for malicious actors. In addition to physical incidents creating physical consequences, exploited cyber vulnerabilities can result in physical consequences, as well.*[3]

These safety and security challenges are not limited to a single sector. Smart grid, intelligent vehicles, next-generation air traffic control, and smart cities are just a few examples of sectors where IoT concepts with new safety and security concerns are being developed and deployed.

> *The inherent level of automation and controllability of positive train control systems makes vulnerabilities particularly dangerous if a malicious actor can exploit them. After obtaining system-level access, an actor could execute a variety of commands, many of which could cause a chain of automated reactions with little or no human oversight.*[4]

Tackling these safety and security challenges requires an approach that embraces highly complex systems at scale and encompasses the full system life cycle from conceptualization

[3]Department of Homeland Security, The Future of Smart Cities: Cyber-Physical Infrastructure Risk. https://ics-cert.us-cert.gov/sites/default/files/documents/OCIA%20-%20The%20Future%20of%20Smart%20Cities%20-%20Cyber-Physical%20Infrastructure%20Risk.pdf, August 2015.

[4]Department of Homeland Security, The Future of Smart Cities: Cyber-Physical Infrastructure Risk, https://ics-cert.us-cert.gov/sites/default/files/documents/OCIA%20-%20The%20Future%20of%20Smart%20Cities%20-%20Cyber-Physical%20Infrastructure%20Risk.pdf.

to realization and assurance. This is the realm of advanced systems engineering and is the theme of this volume. Part I focuses on the fundamentals, describing how systems in an IoT world go beyond the ISO/IEC/IEEE 15288 definition of "a combination of interacting elements to achieve one or more stated purposes" to include those that are aware of, interact with, and shape the world around them. This Part also addresses compositionality—the fact that the properties of an IoT system emerge from the properties of its components and their interactions. For example, the properties of an intelligent transportation system emerge from those of connected vehicles interacting with each other and with intelligent intersections, which are in turn controlled by regional traffic management system, etc. Note that the interacting components at each level in this composition is an IoT application in its own right, a cyber-physical system that is a codesigned hybrid of information and operational technologies (IT and OT) that operates in real time. Analysis of cyber-physical systems—ranging from smart meters and smart phones to continental-scale electric grids and global communications networks—is also addressed in Part I.

Part II provides a series of perspectives on safety and security, starting with the importance of considering the perspective of an attacker in developing a safe and secure design for an IoT system. The perspective of those responsible for producing safe and secure systems is also addressed, with automobile manufacturers as a case study. Cybersecurity as a commercial advantage to a company is also discussed to provide a forward-looking business-model perspective to make up an intelligent transportation system. The assurance perspective—how one may know that a system will do safely and securely what it is designed to do and not do unsafe and insecure things—is also addressed. New perspectives in risk management—dubbed risk engineering—are described that embrace the intricate interdependencies within complex systems that render traditional approaches based on separation of concerns inadequate. Finally the role of standards in providing foundations for interoperability—effective interactions between systems and composability—the ability of systems to serve as components of safe and secure systems-of-systems—is described.

Part III describes application of the concepts in Parts I and II to real-world examples, with cloud computing and smart grid serving as the primary use cases. The first chapter describes combining an attack perspective with concepts from compositionality and risk engineering in designing cloud computing systems that are resilient through effectively managed redundancy, diversity, and reconfigurability. A systems-oriented approach and effective methods for designed-in cybersecurity for cloud computing systems are addressed in the next chapter. The third chapter describes the application of systems engineering and IoT concepts for a safe and secure smart grid. The final chapter describes the development of formal methods and languages for IoT applications, using smart grid as an example.

Collectively, the perspectives set out in this volume provide a foundation for considering the safety and security challenges posed by complex systems in the digital era. Only by meeting these challenges will IoT concepts emerge that can truly enable a world that is safer, more secure, sustainable, livable, and workable.

PART I

SYSTEMS

This page intentionally left blank

EDITOR'S PREFACE

E. Griffor
National Institute of Standards and Technology (NIST), Gaithersburg, MD, United States

A system is a set of interacting components that frequently form a complex whole. Each system has both spatial and temporal boundaries. Systems operate in, are influenced by and influence their environment. Systems can be described structurally, as a set of components and their interactions, or by reference to its purpose. Alternatively, a system can be referenced in terms of its functions and behaviors.

The notion of a system is ubiquitous. It is not simply a technical concept but it lies at the heart of how the mind deals with and conceives of and understands the surrounding world. It is the essence of how we design and build or make things and how we ultimately garner assurance about their behavior. Indeed, the phrase "what we make, makes us" captures a fundamental truth about the relationship between the act of altering our world and how it is we understand that world—we make the world over in the image of our thoughts. *Thought, through sensing and perception and abstraction or conception,* strives to bring order to our experience.

But what of the case where the products of significantly different ways of *thinking* begin to interact? Their interactions are not likely to meet the purposes of any of the designers. What about a world of systems that are allowed to interact, despite the fact that they were not engineered to do so, that they were not intended to do so? This is the world we live in where the Internet provides ubiquitous and unhindered connectivity, possibilities for interaction and composition. Some of the ways these systems interact were *intended* (or *by design*), but so many others were not intended or designed. Sometimes the results are beneficial, but sometimes they have the potential for harm, they are hazardous. The hazards associated with this type of *emergent system behaviors* may result in harm to person and property—this is the topic of system safety. Additionally a system may be vulnerable, may be subject to unauthorized access and modification—this is the topic of system security.

In this preface to the *Handbook of System Safety and Security*, we discuss the concept of a system, system safety and security and review the chapter topics.

1.1 The Need for a Broadly Targeted Handbook of System Safety and Security

The word *system* is *overloaded*, that is, has different meanings to different people. The effort to understand a particular system leads one to ask a few key questions:

- What are the componentor parts of the system?
- What are the interactions between the system's components?
- What are its spatial and temporal boundaries?
- What is its environment?
- What is its structure?
- What function or functions does the system perform?

The interactions between systems, due to the connectivity between systems and to their environment, including human operators, complicate the answers to questions about system safety and security. For example, our need to monitor, measure and control must take into account system connectivity. Hence there is a need to revisit traditional approaches to design for critical concerns such as safety and security. There are also new costs associated with this change in approach. Costs can range from additional component cost, to time delays, to process disruption until new mechanisms are streamlined in. In other words, revisiting these topics must be done from the perspective of all risks.

Though our understanding of systems, as they are rapidly being deployed in our communities and in our nations and across the sectors of the economy, is changing and our approaches to the topics of safety and security are correspondingly diverse, there is a need to begin a broader dialog in order to keep pace with these developments in technology, business, and government. For this reason, the chapters of this *Handbook* reflect the perspectives of experts in each of these sectors. The topics of the chapters are a selection, some technical and others business- and policy-related. It is the hope of the editor, and the contributors, that this volume will serve to inform and stimulate *cross-disciplinary* discussion, study and research on system safety and security.

Part I: Systems

Chapter 1: Editor's Preface and Introduction Edward Griffor

Chapter 1 contains a preface and a brief introduction to the concept of a system (including a discussion of *cyber-physical systems* or CPS), more commonly known as the *Internet of Things* (IoT). CPS are systems that include both logical operations (such as control and feedback) and physical interactions, such as gathering information from the physical realm using sensors or taking an action or actuating that impacts the physical realm. CPS and IoT are the focus of current discussions due to the accelerating deployment of information systems to become the "smarts" of business, industry, government, as well as our cities and nation.

Finally we discuss the concepts of *system safety and security* that treated in this volume and how they relate to one another.

Chapter 2: Composition and Compositionality in CPS—Janos Sztipanovits, Ted Bapty, Zsolt Lattmann, and Sandeep Neema

Chapter 2 introduces composition and compositionality of systems, one of the key challenges to our understanding of systems and of their behaviors. These two notions raise the important questions about how to study and how to gain confidence about the composition of systems.

Cyber-physical systems (CPS) are engineered systems where functionalities and essential properties emerge through the interaction of physical and computational components. One of the key challenges in the engineering of CPS is the integration of heterogeneous concepts, tools, and languages. In order to address these challenges, the authors review a model-integrated development approach for CPS design that is characterized by the pervasive usage of modeling throughout the design process, including application models, platform models, physical system models, environment models, and models of interaction between these modeling aspects. The authors also discuss embedded systems where both the computational processes and the supporting architecture are modeled in a common modeling framework.

Chapter 3: Software Engineering for Model-Based Development by Domain Experts—Monika Bialy, Vera Pantelic, Jason Jaskolka, Alexander Schaap, Lucian Patcas, Mark Lawford, and Alan Wassyng

Chapter 3 discusses the model-based development (MBD) practices that have impacted the development of embedded software in many industries, especially in *safety-critical domains*. The models are typically described using *domain-specific languages and tools* that are readily accessible to domain experts. Domain experts, despite not having formal software engineering training, find themselves creating models from which embedded code is generated and therefore contributing to the design and coding activities of software development. This new role of the domain experts can create new and different dynamics in the interactions with software engineers, and in the development process. In this chapter, the authors describe their experiences as software engineers in multiyear collaborations with domain experts from the automotive industry, who are developing embedded software using the MBD approach. The authors aim to provide guidelines meant to strengthen the collaboration between domain experts and software engineers, in order to improve the quality of embedded software systems, including the safety and security of their systems.

Part II: Perspectives on Safety and Security Chapter 4: Evolving Security—Anuja Sonalker and Edward Griffor

The topic of system security, and in particular that of cybersecurity differs in a critical way from the other concerns we have about systems. Though concerns like safety and resilience do have challenges associated with design, realization, and validation to an ever changing operating environment, security faces an ever evolving adversary. When faced with constantly changing conditions under which a system must continue to deliver its function, designers attempt to model those conditions and test their design against that model. Modeling also becomes important from a measurement standpoint. In order to assess systems and determine their overall risk, their overall security posture, design countermeasures, and then re-assess systems to determine the effectiveness of countermeasures in a provable, reproducible, repeatable quantitative manner, we must be able to model the security, vulnerability, and risk of these systems.

In this chapter the authors introduce new modes of modeling for security adversaries and discuss some basic foundations for adversary modeling. They also discuss how connectivity of systems increases the complexity of system interactions. These complexities also need to be identified and modeled to understand the derivative effect on the overall security posture.

Chapter 5: The Business of Safety—Joseph D. Miller

Chapter 5 discusses system safety from the perspective of system producers. The author illustrates the practice of product or system safety, using the example of system safety in the automobile industry.

Automobiles are some of the most widely deployed, complex systems in our society. While their drivers have a minimal amount of preparation or training to operate them, these systems are growing more complex by the day. Current aspirations are to deploy connected, autonomous vehicles. All involved will face challenges. The title of this chapter "The Business of Safety" is intended to address and discuss several questions, like: What is system safety about? What is it made up of? What do people in this "business" do? What are their fundamental activities and concerns? What do they need to carry on their business? What do they actually produce and how does that relate to the other activities necessary for producing the whole product, other activities necessary for producing the product and addressing other relevant concerns?

Chapter 6: Cybersecurity for Commercial Advantage—James M. Kaplan

Many elements of the work required for a business's offerings are viewed as *noncommercial*, such as cybersecurity. They are regarded by business managers simply as an additional cost that cannot be passed on to customer and that therefore are not recoverable. Many of these elements, and in particular cybersecurity, differ in a critical way from the other concerns that business has. Uneven adoption, including adoption by current or potential business partners, can be a cause of delays in achieving cross-business agreements and can make it much more difficult and costly to achieve and follow your own business's policies regarding those concerns.

In this chapter the author discusses the business of cybersecurity and describes how cybersecurity policies and implementation can be turned into a commercial advantage.

Chapter 7: Reasoning About Safety and Security: The Logic of Assurance—Andrea Piovesean and Edward Griffor

An approach to system safety that emphasizes the work products of the design, verification, and validation activities forces us, in the system's evaluation, to reconstruct the argument and even then there is no standard against which to assess the types of reasoning used. Some constraints on the argumentation are captured in standards that describe how these activities should be performed but only implicitly in the dictates of the standards and not through explicit constraints on the argument itself.

In this chapter we introduce a framework for developing a safety case that clearly distinguishes the part of this reasoning that is common to the analysis of any system and the patterns of acceptable reasoning, identified in standards for specific classes of *cyber-physical systems*. Examples of these prescribed patterns of reasoning can be found in ISO 26262, a standard for automotive software safety and in its predecessors in similar standards in other domains. This framework provides guidance both for the construction of argumentation in a case for system safety and also for assessing the soundness of that sa*fety case*.

Chapter 8: From Risk Management to Risk Engineering: Challenges in Future ICT Systems—Michael Huth, Claire Vishik, and Riccardo Masucci

Information and communications technology (ICT) is an umbrella term that includes any communication device or application, as well as the various services and applications associated with them. Conventional approaches to the design, implementation, and validation of ICT systems deal with one core system concern or two system concerns at a time, for example, the functional correctness or reliability of a system. Additional aspects are often addressed by a separate engineering activity. This *separation of concerns* has led to system engineering practices that are not designed to reflect, detect, or manage the interdependencies of such aspects. For example, the interplay between security and safety in modern car electronics, or between security, privacy, and reliability in connected medical devices.

Current trends and innovation suggest a convergence of disciplines and risk domains in order to deal effectively and predictively with such interdependencies. However, identification and mitigation of composite risks in systems remains a challenge due to the inherent complexity of such interdependencies and the dynamic nature of operating environments.

This environment requires risk management and mitigation be a central and integral part of engineering methods for future systems. In order to address the requirements of the modern computing environment, the authors argue that one needs a new approach to risk, where risk modeling is included in design as its integral part. In this chapter the authors identify some of the key challenges and issues that a vision of risk engineering brings to current engineering practice; notably, issues of risk composition, the multidisciplinary nature of risk, the design, development, and use of risk metrics, and the need for an extensible risk language. This chapter provides an initial view on the foundational mechanisms needed in order to support the vision of risk engineering: risk ontology, risk modeling and composition, and risk language.

Part III: Applications of System Safety and Security

Chapter 9: A Design Methodology for Developing Resilient Cloud Services—Cihan Tunc, Salim Hariri, and Abdella Battou

Cloud Computing is emerging as a new paradigm that aims to deliver computing as a utility. For the cloud computing paradigm to be fully adopted and effectively used, the authors argue that it is critical that the security mechanisms are robust and resilient to malicious faults and attacks. Security in cloud computing is of major concern and a challenging research problem since it involves many interdependent tasks, including application layer firewalls, configuration management, alert monitoring and analysis, source code analysis, and user identity management. It is widely accepted that one cannot build software and computing systems that are free from vulnerabilities and cannot be penetrated or attacked. Therefore it is widely accepted that cyber resilient techniques are the most promising solutions to mitigate cyberattacks and to change the game to the advantage of the defender over the attacker.

Moving Target Defense (MTD) has been proposed as a mechanism to make it extremely difficult for an attacker to exploit existing vulnerabilities by varying the attack surface of the execution environment. By continuously changing the environment (e.g., software versions, programming language, operating system, connectivity, etc.), we can shift the attack surface and, consequently, evade attacks.

In this chapter the authors present a methodology for designing resilient cloud services that is based on *redundancy, diversity, shuffling*, and *autonomic management*. Redundancy is used to tolerate attacks if any redundant version or resource is compromised. Diversity is used to avoid the *software monoculture problem* where one attack vector can successfully attack many instances of the same software module. Shuffling is needed to randomly change the execution environment and is achieved by "hot" shuffling of multiple functionally equivalent, behaviorally different software versions at runtime. The authors also present their experimental results and evaluation of the RCS design methodology. Their experimental results show that their proposed environment is resilient against attacks with less than 7% in overhead time.

Chapter 10: Cloud and Mobile Cloud Architecture, Security and Safety—Charif Mahmoudi

In Chapter 10 the author reviews the notions of *cloud computing* or, more simply, *cloud architecture*. He discusses security and safety as it relates to cloud implementation of systems. This chapter aims to provide guidance about the cloud and the mobile cloud, needed to analyze and make choices, regarding cloud implementation, that are optimal with respect to security and safety constraints. The guidance provided by this chapter can help the software architect to understand the cloud architecture in a manner that will assist in integrating security and safety aspects in an organization's information technology architecture. This chapter targets also technologists, researchers, and scientists; this chapter provides a survey of state-of-the-art techniques, recommendations, and approaches used to make the cloud platform-based systems secure and safe.

In short, the author provides guidance on *cloud architecture for security and safety*. Small and medium businesses, researchers, and government agencies that are planning to implement solutions based on the cloud may find this guidance useful in developing cloud architectures that are suitably adapted to their businesses. The guidance provided will contribute to the success of their cloud implementation even if it is an implementation of a private, hybrid cloud, or an implementation of software components as services in the cloud. Moreover this guidance will assist in ensuring the security and the safety of their implementation.

Chapter 11: A Brief Introduction to Smart Grid Safety and Security—Siham Khoussi and Andreas Mattas

Chapter 11 is intended as a brief introduction to the concepts of the Smart Grid and notions of safety and security for the Smart Grid. It can serve as a guidance for those working within multiple domains related to smart grid and smart grid systems and even for readers interested in understanding what the Smart Grid is, what its basic elements are, and how it differs from the conventional electric power grid. The intended audience includes those working in government, industry, as well as academia in areas related to electric power generation and the environmental aspects of electric power generation.

The authors provide the reader with an overview of the grid and the smart grid architectures, including their component elements and general operation. Based on safety and security paradigms in other domains, the authors highlight some concepts for safety and security of the Smart Grid. Finally the authors provide examples of harm, to both individuals and system assets, that can be caused by not provisioning specific efforts toward understanding system vulnerabilities or hazards. They also give examples of some vulnerabilities and hazards and how they can be addressed in design and operation of the smart grid.

Chapter 12: The Algebra of Systems and System Interactions With an Application to Smart Grid—Charif Mahmoudi, Hasnae Bilil, and Edward Griffor

The existing electric power grid has components for generation, for transmission, and finally for distribution of electric power to large and small users. Power flows from generation components over transmission components to distribution components, servicing large commercial and public facilities, as well as our homes. Growth in demand is responded to by augmenting the grid with additional generation, transmission, and distribution capacity. This enhanced capacity is costly and takes years to provision. Failure to accurately predict growth in demand or inaccurate estimates of grid performance can lead to excessive and unnecessary cost or inadequate capacity. Some have concerns about the impact of less than optimal operation of the power grid and about the impact of continued use of fossil fuels for generation that have increased.

As a result, societal leadership and the public are increasingly aware of alternative approaches to meeting the demand for electric power. As a result, there are current discussions about how one might reshape the electric power "grid" as a "Smart Grid." The proposed changes pose challenges to traditional approaches to grid infrastructure and organization. The "smartness" of the Smart Grid consists in two distinct innovations. The first involves our integrating new technologies into the power grid and the second involves our radically changing the ways that grid elements relate to one another. A Smart Grid manages distributed generation and bidirectional power flow. In the Smart Grid, each new component could potentially affect adversely the performance of other elements of the grid and so one must have a means of expressing and evaluating these proposed grid innovations.

In this chapter the authors propose a language, for expressing the elements of a Smart Grid, and a *composition operator* for composing grid elements. The authors show how this representation of grid elements forms an algebra, under this composition operator, that can facilitate the assessment of architectures for smart grid. They argue that this approach can assist planners and engineers design the Smart Grids of the future and that it can enable planners and engineers to design, and ultimately simulate the composition and the integration of future grid system. This "smart grid algebra" is based on a formal language that offers the expressive power needed to capture the observable behavior and interactions of smart grid components, enables the study of existing smart grid systems, and supports a methodology for the study of critical concerns about the grid such as safety and security.

This page intentionally left blank

COMPOSITION AND COMPOSITIONALITY IN CPS

J. Sztipanovits, T. Bapty, Z. Lattmann, and S. Neema
Vanderbilt University, Nashville, TN, United States

2.1 Introduction

Cyber-physical systems (CPS) are engineered systems where functionalities and essential properties emerge from the networked interaction of physical and computational components. One of the key challenges in the engineering of CPS is the integration of heterogeneous concepts, tools, and languages [1]. In order to address these challenges, a model-integrated development approach for CPS design was advocated by Karsai and Sztipanovits [2], which is characterized by the pervasive use of models throughout the design process, such as application models, platform models, physical system models, environment models, and the interaction models between these modeling aspects. For embedded systems, a similar approach is discussed in Ref. [3], in which both the computational processes as well as the supporting architecture (hardware platform, physical architecture, and operating environment) are modeled within a common modeling framework.

The primary challenge in model-based CPS design flows is improving predictability of system properties "as manufactured" at the end of the design process. A typical characteristic of the current systems' engineering practice is that limited predictability forces the development process to iterate over lengthy design → build/manufacture → integrate → test → redesign cycles until all essential requirements are achieved. There are three fundamental contributors to radically shortening systems development time:

- selecting the *level and scope of abstractions* in the design flow,
- reusing design knowledge captured in component model (CM) libraries and using compositional design methods, and

- introducing extensive automation in the design flow for executing rapid requirements evaluation and design trade-offs.

The most notable example for highly automated model- and component-based design processes is VLSI design supported by electronic design automation tools. While there are arguments that this experience is not portable for a broader category of engineering systems [4], our experience showed that significant improvements can be achieved with the development of horizontal integration platforms for heterogeneous modeling, tool chains, and tool execution [5,6].

The need for establishing horizontal integration platforms for CPS design flows is the consequence of the traditional engineering approach to dealing with heterogeneity and complexity by adopting the "separation of concerns" principle. Sources of heterogeneity in the CPS design space are structured along three dimensions in Fig. 2.1:

- Hierarchical component abstractions that represent CPS designs on different levels of details and fidelity.
- Modeling abstractions that span a wide range of mathematical models such as static models, discrete event models, lumped parameter dynamic models represented as ordinary differential equations, hybrid dynamics, geometry and partial differential equations.
- Physical phenomena including mechanical, electrical, thermal, hydraulic, and other.

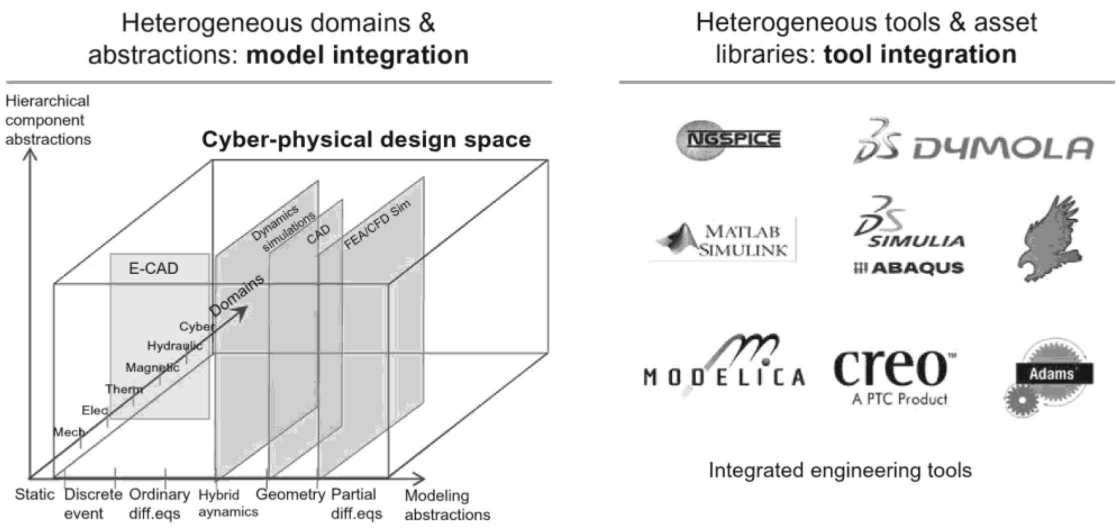

Figure 2.1 Heterogeneity of CPS domains and design tools.

While CPS design requires the exploration of the integrated design space, the separation of concerns principle establishes "slices" in this complex space such as physical dynamics domain, computer-aided design (CAD) domain, electronic CAD (E-CAD) domain, or finite element analysis (FEA) domain. These individual design domains are relatively isolated, linked to different engineering disciplines, and supported by domain-specific tool suites (right side of Fig. 2.1). Since the existing tool suites represent enormous value in terms of design knowledge, established modeling languages, and model libraries, the only reasonable approach to providing support for CPS design flows is to reuse existing assets. This approach works well if the design concerns are independent—but in most cases this is not the case—unless the system is specifically architected for decoupling selected design concerns [1]. Neglecting interdependences across design concerns is one of the primary sources of anomalies and unexpected behaviors detected during system integration. In conclusion, finding solution for the model integration and tool integration challenges are the only practical approach for creating CPS design tool suites.

Heterogeneity of CPS has a significant impact on the central issue of all model- and component-based design methods and on the establishment of a semantically precise composition framework that enable the construction of system models from the models of components. The general requirement for any composition framework is the establishment of composability and compositionality [7]. Composability means that the components preserve their properties in a composed system. Compositionality is achieved if selected essential properties of a system can be computed from the properties of its component. Different engineering disciplines usually have their domain-specific composition frameworks that are synergistic with the modeling abstractions, modeling domains, and properties to be composed. The challenge is to understand how the integration platforms interfere with domain-specific composition and how compositionality can be provided for different properties simultaneously.

In this chapter we discuss some issues of heterogeneous composition for CPS design. The example we use is based on our experience with the development of a model- and component-based design automation tool suite, OpenMETA as part of DARPA's AVM program [8]. The goal of our project was the design, integration, and validation of an end-to-end tool suite for vehicle design. The OpenMETA tool suite [5] gave us opportunity for experimenting with design automation approaches for CPS and for assessing their effectiveness in a sequence of CPS design challenges.

We focus on two issues that are central to model- and component-based design of CPS: model composition and tool composition. First, we show an example of a CPS component model that comprises a suite of domain models with heterogeneous interfaces. The interfaces are designed for supporting domain-specific composition operators and cross-domain interactions. Second, we show that tool integration platforms also bring about composition challenges that interact with model composition. We restrict our discussion to model and tool integration methods for lumped parameter dynamics, which is in itself a complex multifaceted problem.

2.2 Horizontal Integration Platforms in the OpenMETA Tool Suite

Model- and component-based CPS design flows implement a design space exploration process that proceeds from early conceptual design toward detailed design using models and virtual prototyping. This progressive *refinement process* starts with the composition of abstract system models from CMs that capture essential aspects of the system behavior. The system models are evaluated against requirements using simulation and verification tools. The promising designs are refined using higher fidelity CMs and more detailed modeling abstractions. The design process is completed by optimizing relatively few high-fidelity models. The automation of this design process has been a fundamental goal of the OpenMETA tool suite [9–12].

To facilitate the seamless integration of heterogeneous models and tools, OpenMETA complemented the traditional, vertically structured, and isolated model-based tool suites with horizontal integration platforms [13] for models, tools, and executions as shown in Fig. 2.2 [14]. The function of the integration platforms are summarized below.

The modeling functions of the OpenMETA design flow are built on the introduction of the following model types:

1. Component Models (CMs) that include a range of domain models representing various aspects of component properties and behaviors, a set of standard interfaces through which the components can interact and the mapping between the component interfaces and the domain models.
2. Design Models (DMs) that describe system architectures using components and their interconnections.
3. Design Space Models (DSM) that define architectural and parametric variabilities of DMs.

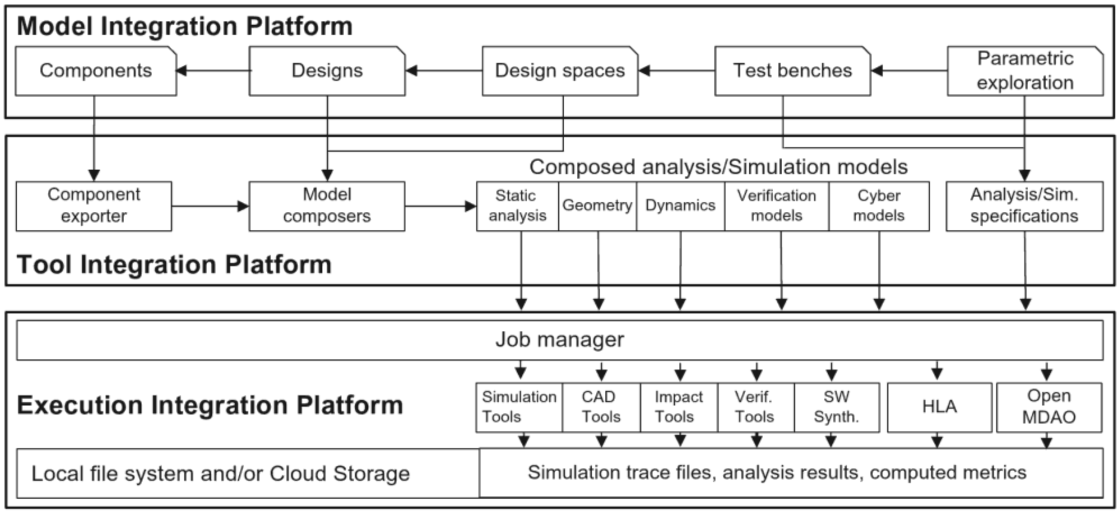
Model Integration Platform
Components
Designs
Design spaces
Test benches
Parametric exploration
Composed analysis/Simulation models
Component exporter
Model composers
Static analysis
Geometry
Dynamics
Verification models
Cyber models
Analysis/Sim. specifications
Tool Integration Platform
Job manager
Execution Integration Platform
Simulation Tools
CAD Tools
Impact Tools
Verif. Tools
SW Synth.
HLA
Open MDAO
Local file system and/or Cloud Storage
Simulation trace files, analysis results, computed metrics

Figure 2.2 Integration platforms.

4. Test Bench Models (TBM) that specify analysis models and analyzes flows for computing key performance parameters linked to specific requirements.
5. Parametric Exploration Models (PEM) that specify regions in the design space to be used for optimizing key performance parameters.

The first three model types focus on the designed system, while the last two represent models of evaluation/optimization processes implemented by test benches. Each test bench contains a link to a system design (the “system under test” object). The system design can be a crude system mock-up composed of low-fidelity CMs at the early stages of the design process, with placeholders for certain subsystems and components whose implementation is not yet clear. Hierarchical DMs define the architecture of a system with its subsystems. Individual designs can be extended to form a design space by adding alternative components and subsystems [15]. The root of a design space has the same interfaces for all design points. Accordingly, even if the number of architectural variants is very large, all associated test benches will remain functional and can be used to evaluate the associated requirements across all point designs generated from the design space. Thus by defining test benches early and executing them periodically, the design space will continually evolve toward containing only satisfying designs.

The fundamental model-integration challenge for OpenMETA is the integration of the five model types described above with the different domain models encapsulated by the components. For example, mobility requirements for a power train, such as "Maximum Speed Hill Climb Sand," are evaluated by a test bench that utilizes lumped parameter dynamic simulation of the power train model with appropriate terrain data. For a given power train architecture, the OpenMETA model composer for lumped parameter dynamics accesses the dynamics models of the individual components in the architecture and composes them into a system model that can be simulated by the Modelica [16] simulation engine. The CMs and the composition mechanism must be flexible enough to enable the use of CMs of different levels of fidelity, even represented in different modeling languages (e.g., Modelica models, Simulink/Stateflow models, Functional Mockup Unit (FMUs), or Bond Graph models) [9]. The TBM links the environment model and the integrated system model to the simulator and creates an executable specification for the evaluation of the "Maximum Speed Hill Climb Sand" performance parameter. Since all design points in the overall design space have the same interface, the TBM can be linked to a design space with many alternative, parameterized architectures. Using the Open MDAO (Multidisciplinary Design Analysis and Optimization) optimization tool, a multiobjective parametric optimization can be performed if the exploration process requires it.

Lumped parameter dynamics and simulation-based evaluation of system designs against mobility requirements is just one example for the many different kinds of test benches required for evaluating alternative powertrain designs. However the general pattern in the overall integration architecture can be clearly seen:

1. *Model Integration Platform*: Heterogeneous models represented in different domain-specific modeling languages are encapsulated in CM libraries. To facilitate model integration, heterogeneous CMs are established with precise composition interfaces and composition operators.
2. *Tool Integration Platform*: Model composers automatically synthesize DMs for test benches by extracting the appropriate CMs from the CM libraries and composing them according to the specification of a candidate architecture. Using models of test benches and parameter exploration processes, analysis flows are integrated for execution on the high-level architecture (HLA) [17] or on Open MDAO.[1]

[1]http://openmdao.org/.

3. *Execution Integration Platform*: Executable TBMs are associated with resources and scheduled for execution on cloud platforms.

Composition of models deposited in the CM libraries, composition of analysis flows inside test benches using simulation and verification tools, and composition of executable analysis images on cloud platforms are in the center of the OpenMETA horizontal integration platforms. In this chapter we restrict our discussion to the selected heterogeneous CM (named AVM component model after the overall program name) and to the composition approach for lumped parameter dynamics.

2.3 AVM Component Model

In a component- and model-based design flow, system models are composed of CMs guided by architecture specifications. To achieve correct-by-construction design, the system models are expected to be heterogeneous multiphysics, multiabstraction, and multifidelity models that also capture cross-domain interactions. Accordingly, the CMs, in order to be useful, need to satisfy the following generic requirements:

1. Elaborating and adopting established, mathematically sound principles for compositionality. Composition frameworks are strongly different in physical dynamics, structure, and computing, which need to precisely defined and integrated.
2. Inclusion of a suite of domain models (e.g., structural, multiphysics lumped parameter dynamics, distributed parameter dynamics, and manufacturability) on an established number of fidelity levels with explicitly represented cross-domain interactions.
3. Precisely defined component interfaces required for heterogeneous composition. The interfaces need to be decoupled from the modeling languages used for capturing domain models. This decoupling ensures independence from the modeling tools selected by the CM developers.
4. Established bounds for composability expressed in terms of operating regimes where the CM remains valid.

These requirements are accepted, but not necessarily practiced in engineering design where component-based approaches are used. A common misconception in physical system modeling is that useful models need to be handcrafted for specific phenomena. One explanation for this is the quite common use of modeling approaches that do not support

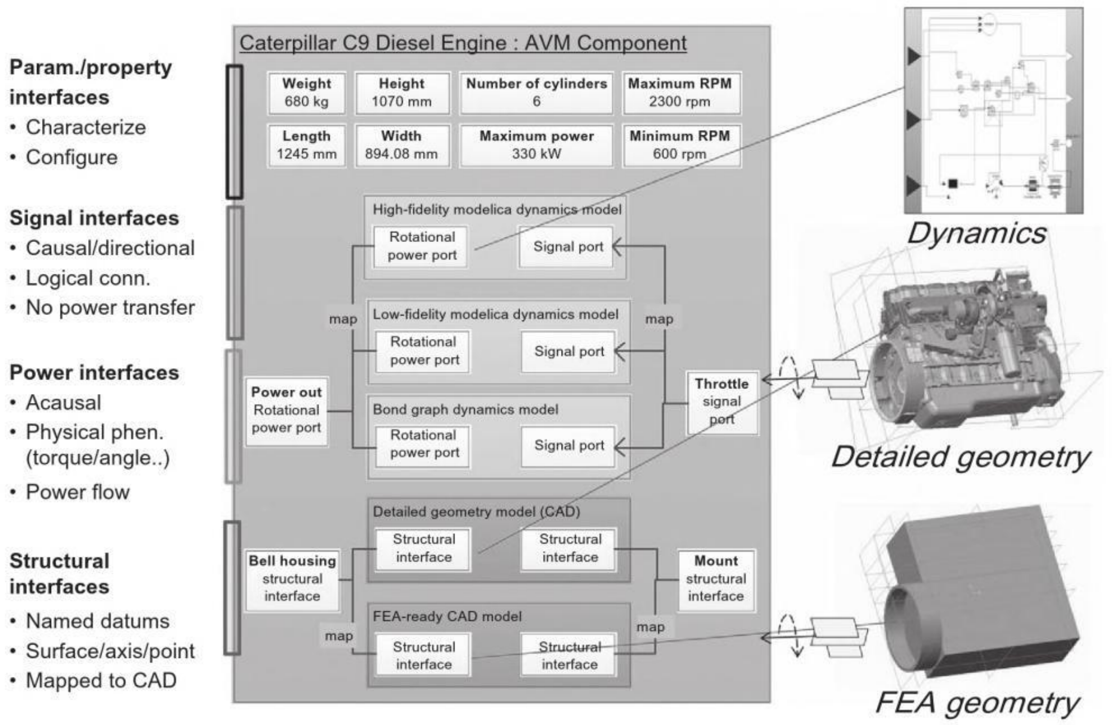

Figure 2.3 Conceptualization of the AVM component model.

compositionality. The AVM component model (Fig. 2.3) placed strong emphasis on compositional semantics that can resolve this problem [18].

A CPS component model must be defined according to the needs of the design process that determines (1) the type of structural and behavioral modeling views required, (2) the type of component interactions to be accounted for, and (3) the type of abstractions that must be utilized during design analytics. We believe that it does not make sense to strive for a "generic" CPS component model, rather, CMs need to be structured to be the simplest that is still sufficient for the goal of "correct-by-construction" design in the given context.

The AVM component model was designed to integrate multi-domain, multiabstraction, and multilanguage structural, behavioral and manufacturing models, and to provide the composition interfaces for the OpenMETA model composers consistently with the needs of power train and hull design [11]. In Fig. 2.3 we illustrate the overall structure of the AVM component model. The main elements of the CM are the followings:

1. The model is a *container of a range of domain models* expressed using domain-specific languages. The actual domain models are referenced from the CMs but stored in separate repositories.
2. Components are characterized by a range of static, physical properties, and labels defined in established ontologies. These static properties are extended with a set of parameters that are changeable during the design process. Fig. 2.3 shows examples for the properties characterizing a Caterpillar C9 Diesel Engine. The static properties and mutable parameters are used in the *early design-space exploration* process [19,20].
3. Lumped parameter physical dynamics plays an essential role in evaluating dynamic behaviors such as mobility properties of designs. Since compositional modeling has been a fundamental goal for us, we chose the *acausal modeling approach* for representing multiphysics dynamics [21]. In this approach dynamics models are represented as continuous time Differential Algebraic Equations (DAE) or hybrid DAEs. Since model libraries may come from different sources, CMs are potentially expressed in different modeling languages such as Bond Graphs (although we dominantly used Modelica-based representations). The multifidelity models are important in assuring scalability in virtual prototyping of systems with a large number of complex components.
4. Models of dynamics implemented computationally inside CPS components are represented using *causal modeling approaches* using modeling languages such as Simulink/Stateflow, ESMoL [22], Functional Mock-up Units [23], or the Modelica Synchronous Library [24].
5. Geometric structure is a fundamental aspect of CPS design. Component geometry expressed as course or detailed CAD models are the basis for deriving geometric features of larger assemblies and performing detailed FEA for a range of physical behaviors (thermal, fluid, hydraulics, vibration, electromagnetic, and others).
6. Modeling and managing cross-domain interactions are in the center of CPS correct-by-construction CPS design. The component modeling language of OpenMETA (described later) includes constructs to define parametric interactions across domain models using formulas.

In constructing an AVM component model from a suite of domain models (such as from Modelica models representing lumped parameter dynamics of physical or computational behaviors, CAD models, models of properties and parameters, and cross-domain interactions) and the mapping of domain modeling elements to component interfaces are time-consuming and error prone. In order to improve productivity, the OpenMETA tools include a full tool suite for importing domain models (such as Modelica dynamic models), integrating them with standard AVM component model interfaces, automatically checking compliance with the standard, and automatically checking model properties, such as restrictions on the types of domain models, well-formedness rules, executability, and others. Based on our direct experience, the automated model curation process resulted in orders-of-magnitude reduction in required user effort for building AVM component model libraries.

In summary, CPS component models are containers of a selected set of domain models capturing those aspects of component structure and behavior that are essential for the design process. While the selected modeling domains are dependent on CPS system categories and design goals, the overall integration platform can still be generic and customizable to a wide range of CPS.

The remaining issue in defining a CPS component model is the specification of component interfaces and the related composition operators.

2.4 Use Case for Semantic Integration

In a heterogeneous multimodeling component approach, component interfaces play a crucial role in making the Model Integration Platform and model composition infrastructure independent from the individual domain-specific modeling languages. This is particularly important, because the different modeling languages (such as Modelica, Simulink, Bond Graphs, CAD, and others) offer internal component and composition concepts that are incompatible with each other and do not match the composition use cases needed in CPS design flows. The design of domain model independent CM interfaces and composition operators must reflect the needs of use cases in the planned design flows.

Due to the complexity and richness of the OpenMETA design flows, we discuss briefly only key elements of the lumped parameter dynamics use cases summarized in Fig. 2.4 [25]. The list of modeling languages used for representing lumped parameter dynamics are shown in the second row in the figure. (TrueTime[2] is a Matlab/Simulink-based simulator for real-time

[2]http://www.control.lth.se/truetime/.

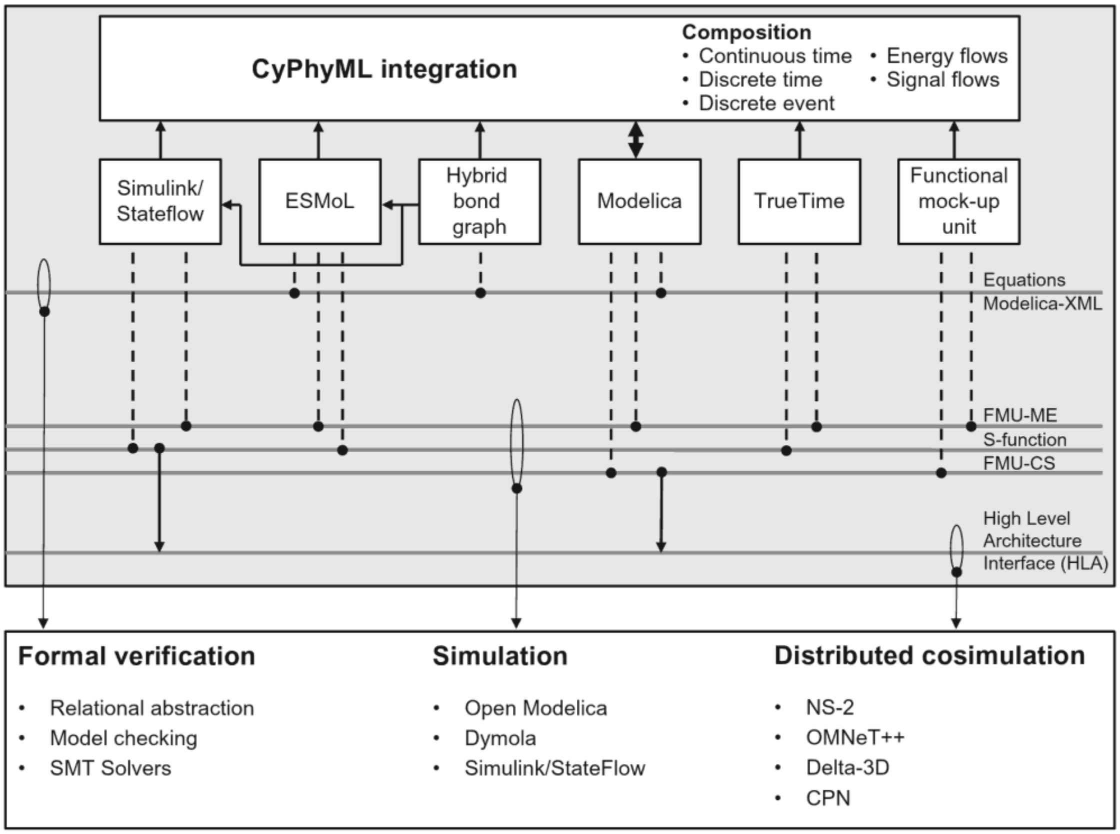

Figure 2.4 Summary of the semantic integration concept for lumped parameter dynamics.

control systems. It enables simulation of controller task execution in real-time kernels, network transmissions in conjunction with continuous plant dynamics.) The modeling languages cover causal (Simulink/StateFlow, ESMoL, TrueTime, and Functional Mock-up Unit) and acausal (Modelica and Hybrid Bond Graph) approaches, continuous, discrete time and discrete event semantics, and facilities for defining physical interactions and signal flows. The connection between the Hybrid Bond Graph language and Simulink/StateFlow and ESMoL represents existing transformation tools from bond graphs to the other languages.

The horizontal bars (Equations, FMU-ME/S-function/FMU-CS, and HLA) represent the target integration domains required by the design flow. Equation-based representations are required by various formal verification tools. The FMU-ME/S-function/FMU-CS bars represent models in the form of input–output computation blocks that can be integrated in simulators. Simulation tools used in OpenMETA include OpenModelica, Dymola, and Simulink/StateFlow. The HLA[3] bar represents the integration domain for distributed cosimulations. OpenMETA uses the HLA standard as distributed simulation integration platform. Cosimulation is effective when single-threaded simulation execution is extremely slow due to the large dynamic range in heterogeneous CPS [17]. Distributed cosimulation is used for virtual prototyping where the simulated system is integrated into and interacts with a complex environment that is simulated by network simulators (OMNeT++ and NS-2), physical environment simulator (Delta-3D), or discrete process simulator (CPN). The vertical dashed lines between the modeling languages and the integration domains represent their relevance for the individual domains. For example, Modelica models (if specified carefully) may contain specification of dynamics in the form of equations that can be exported in Modelica-XML format. In the same time Modelica environments (such as OpenModelica or Dymola) can export models as compiled input–output computation blocks using FMU-ME wrapper or as cosimulation blocks integrated with a solver [23].

The lumped parameter dynamic models encapsulated in AVM components are composed with each other using component interfaces and composition operators. The abstractions describing component interfaces and composition operators are collected in the CyPhyML Model Integration Language (see Fig. 2.4). As the figure suggests CyPhyML is constructed such that the domain-specific languages used for representing component dynamics export a subset of their modeling constructs via a semantic interface. This semantic interface is specified as mapping between the dynamics interface in the AVM component model and abstractions in the different modeling languages. There are two important consequences of introducing the model integration language concept as the cornerstone of semantic integration.

1. Model integration languages (such as CyPhyML) are designed for modeling interactions across domain models. Their semantics is determined by the selected component interfaces and composition operators and not by the domain-specific modeling languages used for specifying embedded CMs (such as Modelica). Accordingly, model integration languages are designed for simplicity. They need to be rich enough for representing cross-domain

[3]https://standards.ieee.org/findstds/standard/1516-2010.html.

interactions, but they can be significantly simpler than the various modeling languages they integrate.

2. Model integration languages evolve as needed. If changing needs of design flows extend to new modeling concepts, new cross-domain interactions, they need to be modified. The most important consequence of the evolving nature of model integration languages is that their semantics need to be formally and explicitly specified to maintain the overall semantic integrity of the multidomain model composition process. This need led to the design and implementation of a Semantic Backplane [10].

The OpenMETA Semantic Backplane [26,27] is at the center of our semantic integration concept. The key idea is to define the structural [28] and behavioral semantics [26,29] of the CyPhyML model integration language using formal metamodeling, and use a tool-supported formal framework for updating the CyPhy metamodels and verifying its overall consistency and completeness as the modeling languages are evolving. The selected tool for formal metamodeling is FORMULA from Microsoft Research [30]. FORMULA's algebraic data types (ADTs) and constraint logic programming (CLP)-based semantics is rich enough for defining mathematically modeling domains, transformations across domains, as well as constraints over domains and transformations.

In Section 2.5 we discuss about component interfaces and composition semantics in OpenMETA, but restrict of the discussion to physical dynamics.

2.5 Component Interfaces and Composition Semantics for Dynamics

As shown in Fig. 2.3, the AVM component model includes four types of interfaces:

1. Parameter/property interface
2. Power interface for physical interactions
3. Signal interface for information flows
4. Structural interface for geometric constraints

Regarding physical interactions, we follow the acausal modeling approach [21]: interactions are nondirectional and there are no input and output ports. Instead, interactions establish simultaneous constraints on the behavior of the connected components by means of variable sharing. For instance, a resistor can be modeled as a two port element, where each port represents a voltage and a current, and the behavior of the resistor is defined by the equations $U1-U2=R^*I1$ and $I1=I2$. In addition to acausal modeling, we adopted the Port-Hamiltonian approach, where physical systems are modeled as network of power-conserving elements like transformers, kinematic pairs, and ideal constraints, together with energy dissipating elements. In this approach, physical components interacting via power ports. These interconnections usually give rise to algebraic constraints between the state space variables of the subsystems leading to a system model which is a mixed set of differential and algebraic equations. The explanation, why such a pair of power variables (effort and flow) is used for describing physical connections, is out of scope in this chapter, but the interested reader can find a great introduction to the topic in Refs. [31,32].

Specification of the CM requires three steps:

1. Specification of the interfaces as typed power ports (electrical power ports, mechanical power ports, hydraulic power ports, and thermal power ports).
2. Specification of the static semantics of the composition by defining constraints over the connection of power ports.
3. Specification of the semantics of connections.

Physical interactions are interpreted over continuous time-domain. (We note here again, that by restricting our discussion to composition of physical interactions, we omit many interesting details regarding the specification of other interactions types and their relationships to each other (e.g., composing causal and acausal models, establishing the link between continuous time and discrete time representations, etc.). For interested readers, these issues are discussed in other papers, such as Refs. [24,26,33–35].

Formally, a component model M from the point of view of dynamic interactions is a tuple M≡{C, A, P, contain, portOf, EP, ES} with the following interpretation:

- C is a set of components,
- A is a set of component assemblies,
- $D=C \cup A$ is the set of design elements,
- P is the union of the following sets of ports: $P_{rotMech}$ is a set of rotational mechanical power ports, $P_{transMech}$ is a set of translational mechanical power ports, $P_{multibody}$ is a set of multi-body power ports, $P_{hydraulic}$ is a set of hydraulic power ports, $P_{thermal}$ is a set of thermal power ports, $P_{electrical}$ is a set of electrical power ports, P_{in} is a set of continuous-time input signal ports, P_{out} is a set of continuous-time output

signal ports. Furthermore, P_P is the union of all the power ports, and P_S is the union of all the signal ports,

- contain : $D \rightarrow A^*$ is a containment function, whose range is $A^* = A \cup \{root\}$, the set of design elements extended with a special root element root,
- portOf : $P \rightarrow D$ is a port containment function, which uniquely determines the container of any port,
- $E_P \subseteq P_P \times P_P$ is the set of power flow connections between power ports,
- $E_S \subseteq P_S \times P_S$ is the set of information flow connections between signal ports.

The specification of the dynamics interface (including both power and signal ports) of the AVM component model using FORMULA ADTs is the following:

```
// Components, component assemblies and design elements
Component ::= new (name: String, id:Integer).
ComponentAssembly ::= new (name: String, id:Integer).
DesignElement ::= Component + ComponentAssembly.
// Components of a component assembly
ComponentAssemblyToCompositionContainment ::=

  (src:ComponentAssembly, dst:DesignElement).

// Power ports
TranslationalPowerPort ::= new (id:Integer).
RotationalPowerPort ::= new (id:Integer).
ThermalPowerPort ::= new (id:Integer).
HydraulicPowerPort ::= new (id:Integer).
ElectricalPowerPort ::= new (id:Integer).
// Signal ports
InputSignalPort ::= new (id:Integer).
OutputSignalPort ::= new (id:Integer).
// Ports of a design element
DesignElementToPortContainment ::= new (src:DesignElement,
dst:Port).
// Union types for ports
Port ::= PowerPortType + SignalPortType.
MechanicalPowerPortType ::= TranslationalPowerPort

               + RotationalPowerPort.

PowerPortType ::= MechanicalPowerPortType +
ThermalPowerPort

             + HydraulicPowerPort
             + ElectricalPowerPort.

SignalPortType ::= InputSignalPort + OutputSignalPort.
// Connections of power and signal ports
```

```
PowerFlow ::=
  new (name:String,src:PowerPortType,dst:PowerPortType,...).
InformationFlow ::=
  new (name:String,src:SignalPortType,dst:SignalPortType,...).
```

The structural semantics for interconnecting dynamics ports are represented as constraints over the connections expressing that the model may not contain any dangling ports, distant connections, or invalid port connections:

```
conforms
no dangling(_),
no distant(_),
no invalidPowerFlow(_),
no invalidInformationFlow(_).
```

For this, we need to define a set of auxiliary rules. Dangling ports are ports that are not connected to any other ports:

```
dangling ::= (Port).
dangling(X) :- X is PowerPortType,
no { P | P is PowerFlow, P.src = X },
no { P | P is PowerFlow, P.dst = X }.
dangling(X) :- X is SignalPortType,
no { I | I is InformationFlow, I.src = X },
no { I | I is InformationFlow, I.dst = X }.
```

A distant connection connects two ports belonging to different components, such that the components have different parents, and neither component is parent of the other one:

```
distant ::= (PowerFlow + InformationFlow).
distant(E) :-E is PowerFlow + InformationFlow,
DesignElementToPortContainment(PX,E.src),
DesignElementToPortContainment(PY,E.dst),
PX != PY,
ComponentAssemblyToCompositionContainment(PX,PPX),
ComponentAssemblyToCompositionContainment(PY,PPY),
PPX != PPY, PPX != PY, PX != PPY.
```

A power flow is valid if it connects power ports of same types:

```
validPowerFlow ::= (PowerFlow).
validPowerFlow(E) :- E is PowerFlow,
X = E.src, X:TranslationalPowerPort,
Y = E.dst, Y:TranslationalPowerPort.
validPowerFlow(E) :- E is PowerFlow,
X = E.src, X:RotationalPowerPort,
Y = E.dst, Y:RotationalPowerPort.
```

```
validPowerFlow(E) :- E is PowerFlow,
X = E.src, X:ThermalPowerPort,
Y = E.dst, Y:ThermalPowerPort.
validPowerFlow(E) :- E is PowerFlow,
X = E.src, X:HydraulicPowerPort,
Y = E.dst, Y:HydraulicPowerPort.
validPowerFlow(E) :- E is PowerFlow,
X = E.src, X:ElectricalPowerPort,
Y = E.dst, Y:ElectricalPowerPort.
```

If a power flow is not valid, it is invalid:

```
invalidPowerFlow ::= (PowerFlow).
invalidPowerFlow(E) :- E is PowerFlow, no validPowerFlow(E).
```

An information flow is invalid if a signal port receives signals from multiple sources, or an input port is the source of an output port:

```
invalidInformationFlow ::= (InformationFlow).
invalidInformationFlow(X) :-X is InformationFlow,
Y is InformationFlow,
X.dst = Y.dst, X.src != Y.src.
invalidInformationFlow(E) :-E is InformationFlow,
X = E.src, X:InputSignalPort,
Y = E.dst, Y:OutputSignalPort.
```

After defining the port types and the structural semantics of the connections, the remaining step in the specification is the semantics of the composition operators (connections). for power flows is represented denotationally through their transitive closure. Using fixed-point logic, we can easily express the transitive closure of connections as the least fixed-point solution for ConnectedPower. Informally, ConnectedPower(x,y) expresses that power ports x and y are interconnected through one or more power port connections:

```
ConnectedPower ::= (src:CyPhyPowerPort, dst:
CyPhyPowerPort).
ConnectedPower(x,y) :-PowerFlow(_,x,y,_,_), x:
CyPhyPowerPort,
y:CyPhyPowerPort;
PowerFlow(_,y,x,_,_), x:CyPhyPowerPort, y:CyPhyPowerPort;
ConnectedPower(x,z), PowerFlow(_,z,y,_,_), y:
CyPhyPowerPort;
ConnectedPower(x,z), PowerFlow(_,y,z,_,_), y:
CyPhyPowerPort.
```

More precisely, Px = {y | ConnectedPower(x, y)} is the set of power ports reachable from power port x. The behavioral semantics of power port connections is defined by a pair of equations generalizing the Kirchoff equations. Their form is the following:

$$\forall \mathrm{x} \in \mathrm{CyPhyPowerPort} \cdot \left(\sum_{\mathrm{y} \in \{\mathrm{y} | \mathrm{ConnectedPower} \quad (\mathrm{x},\mathrm{y})\}} e_{\mathrm{y}} = 0 \right)$$

$$\forall \mathrm{x}, \mathrm{y}(\mathrm{ConnectedPower}(x, y) \rightarrow e_{\mathrm{x}} = e_{\mathrm{y}})$$

We can formalize this FORMULA in the following way:

```
P : ConnectedPower → eq + addend.
P [[ConnectedPower]] =

    eq(sum("CyPhyML_powerflow",flow1.id), 0)
    addend(sum("CyPhyML_powerflow",flow1.id), flow1)
    addend(sum("CyPhyML_powerflow",flow1.id), flow2)
    eq(effort1, effort2)
```

where

```
    x = ConnectedPower.src, y = ConnectedPower.dst, x != y,
    DesignElementToPortContainment(cx,x), cx:Component,
    DesignElementToPortContainment(cy,y), cy:Component,
    PP [[x]] = (effort1,flow1),
    PP [[y]] = (effort2,flow2).
```

The specifications above are only short illustrations of the nature and scope of the full formal specification of the AVM component model and the CyPhyML Model Integration Language. Together with the specification of the model composers, the size of the Semantic Backplane is nearly 20K line of FORMULA code. It is our experience that development and consistent application of the specification frameworks was key in keeping the OpenMETA model and tool integration components consistent.

2.6 Formalization of the Semantic Interface for Modeling Languages

So far, we formally defined the semantics of the compositional elements of CyPhyML but we have not specified the semantic interface between the domain-specific modeling languages such as Modelica, Simulink/StateFlow, Bond Graph Language, ESMoL, and CyPhyML. Note that we can easily add other languages to the list following the same steps as presented here. We show the specification of semantic interface only for Modelica.

Modelica is an equation-based object-oriented language used for systems modeling and simulation. Modelica supports component-based development through its model and connector concepts. Models are components with internal behavior and a set of ports called connectors. Models are interconnected by connecting their connector interfaces. A Modelica connector is a set of variables (input, output, acausal flow or potential, etc.) and the connection of different connectors define relations over their variables. In the following we discuss the integration of a restricted set of Modelica models in CyPhyML: we consider models that contain connectors that consist of either exactly one input/output variable, or a pair of effort and flow variables.

The semantics of Modelica power ports are explained by mapping to pairs of continuous time variables:

```
MPP : ModelicaPowerPort → cvar,cvar.
MPP [[ModelicaPowerPort]] =

   (cvar("Modelica_potential",ModelicaPowerPort.id),
   cvar("Modelica_flow",ModelicaPowerPort.id)).
```

The semantics of Modelica signal ports is explained by mapping to continuous time variables:

```
MSP : ModelicaSignalPort → cvar.
MSP [[ModelicaSignalPort]] =
cvar("Modelica_signal",ModelicaSignalPort.id).
```

The semantics of Modelica and CyPhyML power port mappings is equality of the power variables. Formally,

```
MP : ModelicaPowerPortMap → eq.
MP [[ModelicaPowerPortMap]] =

   eq(cyphyEffort, modelicaEffort)
   eq(cyphyFlow, modelicaFlow)

where

   modelicaPort = ModelicaPowerPortMap.src,
   cyphyPort = ModelicaPowerPortMap.dst,
   PP [[cyphyPort]] = (cyphyEffort, cyphyFlow),
   MPP [[modelicaPort]] = (modelicaEffort, modelicaFlow).
```

The semantics of Modelica and CyPhyML signal port mappings is equality of the signal variables.

```
MS : ModelicaSignalPortMap → eq.
MS [[ModelicaSignalPortMap]] = eq(MSP

   [[ModelicaSignalPortMap.src]],
   SP [[ModelicaSignalPortMap.dst]]).
```

An interesting aspect of the specification of semantic interface between CyPhyML and the domain-specific modeling languages is the assignments of physical units for power ports. Each PortUnit assigns two units to each power port: one to its effort variable and one to its flow variable:

```
PortUnit ::= [port:PowerPort ⇒ effort:Units, flow:Units].
PortUnit(x,"V","A") :- x is ElectricalPowerPort;

   x is ElectricalPin;
   x is ElectricalPort.

PortUnit(x,"m","N") :- x is TranslationalPowerPort;

   x is TranslationalFlange.

PortUnit(x,"N","m/s") :- x is MechanicalDPort.
PortUnit(x,"rad","N.m") :- x is RotationalPowerPort;

   x is RotationalFlange.

PortUnit(x,"N.m","rad/s") :- x is MechanicalRPort.
PortUnit(x,"kg/s","Pa") :- x is HydraulicPowerPort;

   x is FluidPort;
   x is HydraulicPort.

PortUnit(x,"K","W") :- x is ThermalPowerPort;

   x is HeatPort;
   x is ThermalPort.

PortUnit(x,"NA","NA") :- x is MultibodyFramePowerPort.
PortUnit(x,"Pa,J/kg","kg/s,W") :- x is FlowPort.
```

2.7 Conclusion

We have presented an example for establishing aspects of composition and compositionality in a CPS design flow. After deciding the goal of the composition, the required steps are generic: we need to establish a CM, define interfaces, define composition operators, and make a mapping between the modeling describing the component behavior and the modeling language representing the composed system. Although we did not cover many aspects and details of composition we developed in the AVM project, the example is sufficient for illustrating some general conclusions:

1. We believe that CPS design problems require different kinds of CMs and composition methods. Components are containers of relevant and reusable design knowledge represented in domain-specific languages. The selection of model types

need to be matched with the CPS category and the type of analyses required during the design process. It is not the particular combination of domain models are generalizable, but the fact that the formation of CPS component models require cross-domain modeling and model integration. This insight led us to construct a reusable Model Integration Platform that includes methods, tools, and libraries for creating model integration languages, specifying their formal semantics, and structuring those in a Semantic Backplane that provides foundation for CPS composition frameworks in highly different application domains. The OpenMETA Semantic Backplane is at the center of our semantic integration concept. The key idea is to define the semantics of the CyPhyML model integration language using formal metamodeling, and to use a tool-supported formal framework for updating the CyPhyML metamodels and verifying its overall consistency and completeness as the modeling languages are evolving. The selected tool for formal metamodeling is FORMULA [36] from Microsoft Research. FORMULA's ADTs and CLP-based semantics are effective at mathematically defining modeling domains, transformations [37] across domains, as well as constraints over domains and transformations. At the conclusion of the AVM project, the OpenMETA Semantic Backplane included the formal specification of CyPhyML, the semantic interfaces to all constituent modeling languages, and all model transformations used in the tool integration framework. (The size of the specifications is 19,696 lines out of which 11,560 are autogenerated and 8136 are manually written.)

2. Composition occurs in several semantic domains in CPS design flows even inside a single analysis thread. For example, the system level Modelica model for a power train using the composition semantics described above yields a large number of equations for which the simulation with a single Modelica simulator may be extremely slow. In this case we may take the composed system level model and decompose it again, but not along the component/subsystem boundaries but along physical phenomena (mechanical processes and thermal process) so that we can separate the fast and slow dynamics [17]. This decompositions leads to two models that can be cosimulated using the HLA cosimulation platform (see Fig. 2.4), so the recomposition of the system level model occurs in a different semantic domain.
3. In a naïve view, model and tool integration is considered to be an interoperability issue between multiple models that

can be managed with appropriate syntactic standards and conversions. In complex design problems, these approaches inevitably fail due to the rapid loss of control over the semantic integrity of the diverse set of models involved in real-design flows. The "cost" of introducing a dynamic, evolvable model integration language is that mathematically precise formal semantics for model integration had to be developed under OpenMETA.

4. The dominant challenge in developing OpenMETA was integration: models, tools, and executions. The OpenMETA integration platforms included ~1.5M lines of code that is reusable in many CPS design context. In the AVM project, OpenMETA integrated 29 open source and 8 commercial tools representing a code base which is estimated 2 orders of magnitude larger than OpenMETA [6]. The conclusion is that integration does matter. It is scientifically challenging and yields major benefits. This is particularly true in design automation for CPS, where integrated design flows are still not reality.

Acknowledgments

The reported results are conclusions of research partly supported by the Defense Advanced Research Project Agency (DARPA) under award #HR0011-12-C-0008 and #N66001-15-C-4033, the National Science Foundation under award #CNS-1035655 and #CNS-1238959, and NIST Award #NIST 70-NANB15H312.

References

[1] J. Sztipanovits, X. Koutsoukos, G. Karsai, N. Kottenstette, P. Antsaklis, V. Gupta, et al., Toward a science of cyber-physical system integration, Proc. IEEE 100 (1) (2012) 29–44. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6008519>.

[2] G. Karsai, J. Sztipanovits, Model-integrated development of cyber-physical systems, Software Technologies for Embedded and Ubiquitous Systems, Springer, 2008, pp. 46–54. <http://link.springer.com/chapter/10.1007/978-3-540-87785-1_5>.

[3] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty, Model-integrated development of embedded software, Proc. IEEE 91 (1) (2003) 145–164. <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1173205>.

[4] D.E. Whitney, Physical limits of modularity, MIT Working Paper Series ESD-WP-2003-01.03-ESD Internal Symposium, 2003.

[5] J. Sztipanovits, T. Bapty, S. Neema, X. Koutsoukos, E. Jackson, Design tool chain for cyber physical systems: lessons learned, in: Proceedings of DAC'15, DAC'15, 07–11 June 2015, San Francisco, CA, USA.

[6] J. Sztipanovits, T. Bapty, S. Neema, X. Koutsoukos, J. Scott, The METa Toolchain: Accomplishments and Open Challenges, No. ISIS-15-102, 2015 (Google Scholar Download: The META Toolchain_Accomplishments and Open Challenges.pdf).
[7] G. Gossler, J. Sifakis, Composition for component-based modeling, Sci. Comput. Program. 55 (1–3) (2005).
[8] P. Eremenko, Philosophical Underpinnings of Adaptive Vehicle Make, DARPA-BAA-12-15. Appendix 1, December 5, 2011.
[9] Zs. Lattmann, A. Nagel, J. Scott, K. Smyth, C. van Buskirk, J. Porter, et al., Towards automated evaluation of vehicle dynamics in system-level design, in: Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2012, 12–15 August 2012, Chicago, IL.
[10] G. Simko, T. Levendovszky, S. Neema, E. Jackson, T. Bapty, J. Porter, J. Sztipanovits, Foundation for model integration: semantic backplane, in: Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2012, 12–15 August 2012, Chicago, IL.
[11] R. Wrenn, A. Nagel, R. Owens, D. Yao, H. Neema, F. Shi, K. Smyth, Towards automated exploration and assembly of vehicle design models, in: Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2012, 12–15 August 2012, Chicago, IL.
[12] O.L. de Weck, Feasibility of a 5× speedup in system development due to meta design, in: 32nd ASME Computers and Information in Engineering Conference, August 2012, pp. 1105–1110.
[13] J. Sztipanovits, T. Bapty, S. Neema, L. Howard, E. Jackson, OpenMETA: a model and component-based design tool chain for cyber-physical systems, in: S. Bensalem, Y. Lakhneck, A. Legay (Eds.), From Programs to Systems—The Systems Perspective in Computing, LNCS, vol. 8415, Springer-Verlag, Berlin Heidelberg, 2014, pp. 235–249.
[14] J. Sztipanovits, Model integrated design tool suite for CPS, Invited Talk at University of Hawaii, Honolulu, 9 April 2015 (Figure 2).
[15] H. Neema, S. Neema, T. Bapty, Architecture Exploration in the META Tool Chain, ISIS-15-105, Technical Report, ISIS/Vanderbilt University, 2015.
[16] Modelica Association, Modelica—A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification, Version 3.2. <www.modelica.org/documentas/ModelicaSpec32.pdf>, March 24, 2010.
[17] H. Neema, J. Gohl, Z. Lattmann, J. Sztipanovits, G. Karsai, S. Neema, et al. Model-based integration platform for FMI co-simulation and heterogeneous simulations of cyber-physical systems, in: Proceedings of the 10th International Modelica Conference, Lund, Sweden, 10–12 March 2014, pp. 235–245.
[18] T. Bapty, OpenMETA Project Overview, Project Briefing, March 2012 (Figure 3).
[19] H. Neema, Z. Lattmann, P. Meijer, J. Klingler, S. Neema, T. Bapty, et al., Design space exploration and manipulation for cyber physical systems, in: IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems (IDEAL' 2014), Springer-Verlag Berlin Heidelberg, 2014.
[20] E. Jackson, G. Simko, J. Sztipanovits, Diversely enumerating system-level architectures, in: Proceedings of EMSOFT 2013, Embedded Systems Week, September 29–October 4, 2013, Montreal, CA.

[21] J.C. Willems, The behavioral approach to open and interconnected systems, IEEE Control Systems Magazine, December 2007, pp. 46–99.
[22] J. Porter, G. Hemingway, H. Nine, C. van Buskirk, N. Kottenstette, G. Karsai, J. Sztipanovits, The ESMoL language and tools for high-confidence distributed control systems design. Part 1: Design Language, Modeling Framework, and Analysis. Tech. Report ISIS-10-109, ISIS, Vanderbilt Univ., Nashville, TN, 2010.
[23] Functional Mock-up Interface. <www.fmi-standard.org>.
[24] Modelica Association, Modelica Language Specification Version 3.3. Revision 1. <https://www.modelica.org/documents/ModelicaSpec33Revision1.pdf>, 11July 2014.
[25] J. Sztipanovits, Model Integration Challenge in Cyber Physical Systems, Tutorial, NIST, 19 January 2012 (Figure 4).
[26] G. Simko, J. Sztipanovits, Model integration challenges in CPS, in: R. Rajkumar (Ed.), Cyber Physical Systems, Addison-Wesley, 2015.
[27] G. Simko, T. Levendovszky, M. Maroti, J. Sztipanovits, Towards a theory for cyber-physical systems modeling, in: Proc. 3rd Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy'13), 08–11 April 2013, Philadelphia, PA, USA, pp. 1–6.
[28] E. Jackson, J. Sztipanovits, Formalizing the structural semantics of domain-specific modeling languages, J. Softw. Syst. Model. (September 2009) 451–478.
[29] K. Chen, J. Sztipanovits, S. Neema, Compositional specification of behavioral semantics, in: R. Lauwereins, J. Madsen (Eds.), Design, Automation, and Test in Europe: The Most Influential Papers of 10 Years DATE, Springer, 2008.
[30] E.K. Jackson, T. Levendovszky, D. Balasubramanian, Reasoning about metamodeling with formal specifications and automatic proofs, in: J. Whittle, T. Clark, T. Kühne (Eds.), Model Driven Engineering Languages and Systems, vol. 6981, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 653–667.
[31] A. van der Schaft, D. Jeltsema, Port-Hamiltonian systems theory: an introductory overview, Found. Trends Syst. Control 1 (2–3) (2014) 173–378. Available from: http://dx.doi.org/10.1561/2600000002.
[32] D. Karnopp, D.L. Margolis, R.C. Rosenberg, System Dynamics Modeling, Simulation, and Control of Mechatronic Systems, John Wiley & Sons, Hoboken, NJ, 2012.
[33] G. Simko, D. Lindecker, T. Levendovszky, E.K. Jackson, S. Neema, J. Sztipanovits, A framework for unambiguous and extensible specification of dsmls for cyber-physical systems, in: Engineering of Computer Based Systems (ECBS), 20th IEEE International Conference and Workshops on the, IEEE, 2013, pp. 30–39.
[34] D. Lindecker, G. Simko, I. Madari, T. Levendovszky, J. Sztipanovits, Multi-way semantic specification of domain-specific modeling languages, in: Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the, IEEE, April 2013, pp. 20–29.
[35] G. Simko, D. Lindecker, T. Levendovszky, S. Neema, J. Sztipanovits, Specification of cyber-physical components with formal semantics–integration and composition, in: Model-Driven Engineering Languages and Systems, MODELS'2013, Springer Berlin Heidelberg, 2013, pp. 471–487.
[36] <http://research.microsoft.com/formula>.
[37] D. Lindecker, G. Simko, T. Levendovszky, I. Madari, J. Sztipanovits, Validating transformations for semantic anchoring, J. Object Technol. 14 (3) (August 2015), pp. 2:1-25, <http://dx.doi.org/10.5381/jot.2015.14.3.a2>.

SOFTWARE ENGINEERING FOR MODEL-BASED DEVELOPMENT BY DOMAIN EXPERTS

M. Bialy[1], V. Pantelic[1], J. Jaskolka[2], A. Schaap[1], L. Patcas[1], M. Lawford[1], and A. Wassyng[1]

[1]McMaster University, Hamilton, ON, Canada [2]Stanford University, Stanford, CA, United States

3.1 Introduction and Motivation

Early in the computer age, it was recognized that an ad hoc programming approach was not suitable for developing nontrivial software systems. In the words of a famous computer scientist, Edsger Dijkstra: “To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.” Therefore, a systematic engineering approach including planning, problem understanding, requirements gathering and specification, design, programming, and verification became necessary. This is how *software engineering* was born. According to ISO/IEC/IEEE Standard 24765 [1], software engineering is defined as, “The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software.”

Unfortunately, decades later, software development and maintenance is still not practiced with the same discipline exercised in other engineering fields. Developing software is often deemed trivial by nonpractitioners. This perception is mostly due to software’s *malleability.* Since software itself is not physical, a modification to software is considered “merely a code change.” This perception, however, is wrong. Experience teaches us that software should be modified with the same rigor as any other engineering product, e.g., an engine, power inverter, or airplane brakes. The effect of a change should be evaluated on a design first, and then thoroughly verified. This is an approach especially necessary in modern systems, which increasingly rely on software. Software accounts for 80% of military aircraft functions [2] and 80% of innovations in vehicles [3]. Software has also grown to be a significant source of accidents and product recalls [4]. Moreover, numerous examples of software-related accidents span the safety-critical domains of aerospace [5], medical [6], and automotive [7], with many more examples listed in [8,9]. For such safety-critical systems, errors can potentially result in loss of life, environmental damage, and/or major financial loss. Therefore, practicing software engineering with the same rigor and discipline recognized in other areas of engineering is crucial to the successful development and safe operation of modern software-intensive systems.

Model-Based Development (MBD) has become a predominant paradigm in the development of embedded systems across industries, including aerospace, automotive, and nuclear. This is mostly due to its appeal of automatic code generation from models, early verification and validation, and rapid prototyping. Furthermore, domain-specific modeling languages used in MBD are easily learned and used by domain experts (experts in the field of the application), allowing them to design, generate code, and verify their own algorithms, using familiar terminology and abstractions. Therefore, the MBD paradigm assigns domain experts a different role than the one they typically have in a traditional software development process. However, domain experts have backgrounds in mechanical engineering, electrical engineering, or other related fields, but typically have no formal education in software engineering. For example, many leading Japanese software specialists believe the majority of Japanese software developers have not been formally educated in software engineering [10].

Our work builds on experience drawn from collaborations between our team of software engineers and domain experts in the automotive industry. While working on multiyear projects with automotive Original Equipment Manufacturers (OEMs), we have interfaced with a number of domain experts from both academia and industry.

First, we have witnessed a large difference in terminology used by software engineers and automotive domain experts[1]. We (partially) address this communication gap between the two communities by explaining the terminology originating in

[1]In fact, the term *domain experts* is widely known and used within the software engineering community, while domain experts themselves are largely unaware of the term.

software engineering that is commonly used in development of embedded systems.

Second, domain experts use and/or help develop various software artefacts, often without a clear picture of their intent and their ultimate effect on the quality of software.

This chapter clarifies the role of some of the most commonly used (and those that are not, but should be) software engineering principles, practices, and artefacts by viewing them from a software engineering perspective, and presenting how they affect software correctness, safety, and other software qualities. Therefore, this chapter aims at strengthening the collaboration between software engineers and domain experts, by offering domain experts a high-level understanding of software engineering practices and artefacts, enabling their more effective use. In the process, a number of MBD misconceptions and limitations are addressed. Further, we discuss issues in the industrial practice of MBD, and suggest solutions whenever possible, or point to avenues for research to address issues for which a solution currently does not exist. The chapter is focused on the development of embedded software using Matlab Simulink, the de facto standard in model-based design of embedded systems. Ironically, Simulink itself neglects some major software engineering principles, and this issue is also discussed in this chapter. While the focus of this chapter is on the MBD of embedded systems using Matlab Simulink, many of the discussions are applicable to software engineering in general. Therefore, we view this chapter as a useful tutorial primarily for domain experts involved in the development of software intensive systems, but also for software practitioners in general, managers in related fields, and any staff involved in software and/or software development.

The remainder of this chapter is organized as follows. Section 3.2 describes the overall MBD software engineering process and serves as a prelude to the subsequent sections. The subsequent sections, Sections 3.3, 3.4, 3.5, and 3.6, then provide insight into commonly encountered questions and misconceptions in industry regarding requirements, design, implementation, and verification and validation, respectively. Finally, Section 3.7 presents conclusions and directions for future work.

3.2 Development Process: How Do You Engineer Software?

Software is not only code, and developing software is not just programming. Software includes requirements, design, test

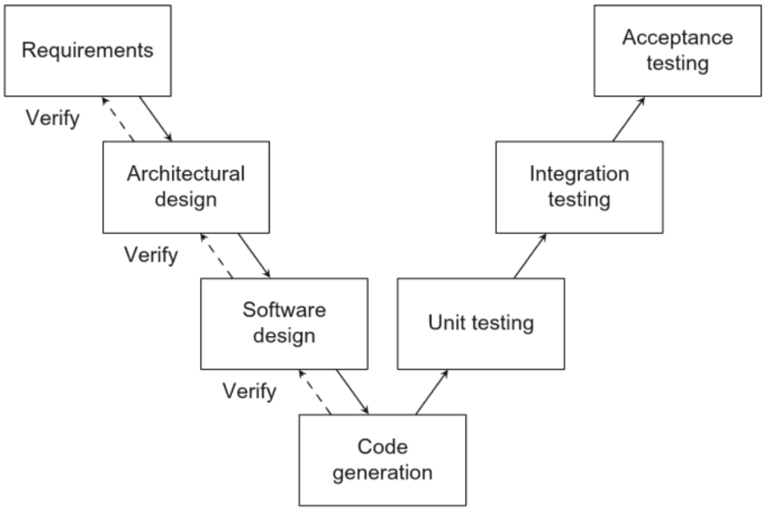

Figure 3.1 V-Model development process for model-based development (MBD).

reports, and other documentation which are artefacts resulting from the different phases in the engineering process. As with all engineering disciplines, well-defined processes must be followed in order to construct quality systems which operate safely. The most common description of the software engineering process within the MBD of embedded software is known as the V-model, shown in Fig. 3.1. Although many process models exist for software systems, the V-model is the most widely accepted model for embedded safety-critical systems because of its focus on testing at different levels. Moreover, standards such as the automotive standard ISO 26262 [11] prescribe its use. In this section, we provide a summary of the phases of the V-model which are further elaborated in the sections that follow.

3.2.1 What Are the Phases of the Engineering Process? How Are Domain Experts Involved?

The development process begins with the gathering and specification of requirements. In this phase, a high-level description of *what* the system should do is determined, without providing any details as to *how* it is done. As a result of the requirements phase, a software requirements specification (SRS) is produced and agreed upon in order to act as a contract between stakeholders and developers, that is, a mutual agreement of the expectations from the system. This phase typically involves close collaboration between software engineers, analysts, managers, with domain experts providing technical breadth and depth within their respective domains. For example, our experience is that a separate team of safety experts plays an integral role in contributing to the development and analysis of safety requirements for automotive systems.

Once a working set of requirements for the system has been established, a high-level architectural design is planned. The architectural design should strive to integrate principles of software engineering (e.g., *modularity* and *encapsulation*), that will be further explained in Section 3.4, in order to minimize complexity and facilitate component reusability. Again, managers, software engineers, and domain experts are primarily involved at this stage, with third-party suppliers also participating where necessary. Architectural design is then verified by way of reviews, simulations (if the corresponding executable specification exists), etc. Next, a software solution that satisfies the requirements and conforms to the architectural design is developed. In MBD, this is largely done by constructing models in accordance with language guidelines and standards. This phase includes defining the necessary component modules, algorithms, data structures, and other detailed design elements necessary for the implementation (or in the case of MBD, code generation). In practice, one or more components or modules are assigned to an individual to "own," that is, to develop and maintain. In current MBD practice, we have found that domain experts design software and rapidly prototype designs, which are later transferred to other engineers to prepare for production as well as maintain. Ideally, these software development activities should be performed by software engineers. They will be well-versed in implementing software using accepted engineering best practices and principles. Close collaboration with a domain expert, knowledgeable about the domain-specific context, will provide guidance toward a solution.

A major benefit of the MBD approach is the ability to automatically generate the implementation code from design models. This significantly reduces implementation errors and development time when compared to traditional programming [12], and also enables domain experts' deep involvement in the development process. The same component "owners" responsible for designing the software will generate its corresponding code. If needed, another separate team of engineers may be responsible for code generation rule customization, which typically comes from the recommendations and suggestions of domain experts. After generating an implementation of the software system, verification takes place to ensure that the system that is implemented is the one that was designed and expected. MBD offers the ability to perform tests early in the development cycle, at different levels, before the software even makes it onto the hardware. There are various stages of testing which occur throughout the development process. For example, *unit testing* verifies each software component individually and independently from the rest of the system, whereas *integration testing* combines software components to verify the system as a whole, and *acceptance testing* verifies that the system satisfies its requirements and performs as expected. In general, the embedded system under development is modeled as a controller, which aims to control some physical system using supervisory logic. The physical system is described in a plant model, which provides the controller with inputs. Depending on the development stage of the controller and the platform upon which the plant is simulated, different testing strategies can be utilized throughout the MBD process:

Model-in-the-Loop (MiL): The controller and plant models are simulated in their development environment (e.g., Simulink).

Software-in-the-Loop (SiL): The controller embedded code, generated from the model into hardware-dependent code, is simulated with the plant model, both on the same machine, typically on PC hardware.

Processor-in-the-Loop (PiL): The controller embedded code is loaded onto the embedded processor (hardware), and is simulated with the plant model in real time.

Hardware-in-the-Loop (HiL): The controller embedded code is run on the final hardware, an electronic control unit (ECU), with a simulated plant model in real time.

The phase following software release is *maintenance* (not depicted in Fig. 3.1), where either defects are fixed or software is modified to satisfy new requirements. In fact, ease of maintenance (maintainability) is one of the very important qualities of software, that, although often not explicitly required, motivates many of the activities in the development process from Fig. 3.1. Software is maintained through collaborative efforts between domain experts and software engineers. For example, in some companies, a software engineer will be in charge of maintaining a software feature (Simulink model). The software will be modified in collaboration with a domain expert, typically in charge of several similar features (Simulink models).

3.2.2 How Important Are the Tools?

Appropriate tool support in each phase of the process by way of a comprehensive tool-chain that facilitates different activities, including change management, build management, bug tracking, etc., is crucial for the success of a development process [13]. Engineering a system often requires many iterations of the development process and its phases. For example, as the software design is developed, requirements can change, making it necessary to go back and repeat the requirements phase. In fast-paced industries such as automotive, performing such iterations quickly is greatly facilitated through the use of tool-chains which span the entire process, and can fully- or semiautomate designing and implementing changes.

3.2.3 An Illustrative Example: Transmission Control Software

For the purpose of illustrating and highlighting the software engineering process for MBD described in the remainder of this chapter, we will consider a small automotive example that was provided by one of our industrial partners, as presented in [14]. Suppose that we need to design and develop the embedded software to control the automatic transmission system of a hybrid-electric vehicle based on requests made by the driver to change gears between park (P), reverse (R), neutral (N), and drive (D) via a "PRND" shifter, typically in the form of a lever or knob within the vehicle console. When using the vehicle, a driver makes requests to change the transmission gear via the shifter (e.g., switch from park to drive), at which point the embedded software needs to decide whether or not to grant the driver's request based on a number of system conditions, such as faults and the availability of certain components. In the subsequent sections of this chapter, we will use this simple illustrative example to demonstrate how to specify software requirements, to translate those requirements into suitable model-based designs, and to verify and validate that the implemented design exhibits the expected system behavior.

3.3 Requirements: What Should Your Software Do?

3.3.1 How Important Are Good Requirements?

Contrary to common belief, software rarely fails. More often than not, the software behaved *exactly* as it was required to, but it was the requirements that were flawed [15]. Some sources assert that over 90% of software issues result from deficient requirements, leaving merely 10% of issues to be caused by design and coding problems [16]. Therefore, experience teaches us that getting requirements right as well as precisely specifying them is essential for the establishment of safe and effective systems [17]. The terms "requirements" and "requirements specification" are taken from software engineering, and are not a part of domain experts' jargon. Our experience shows that domain experts would rather refer to it as "specification" or "spec" only.

3.3.2 What Is the Purpose of a Requirements Specification? Who Uses It?

Before building a safe and usable system, an understanding of what it is meant to accomplish and what qualities it should possess is required. Requirements specify *what* the system should do, and a SRS is an artefact in which software requirements are documented and maintained. A requirements specification acts as a contract between users and software developers. It is also used by verifiers to show that the software satisfies its requirements and by managers to estimate and plan for resources. In our experience, the requirements specification is essential for helping mitigate the impact of developer turnover, especially within the automotive industry which experiences frequent movement of personnel.

3.3.3 Simulink Models Are NOT Requirements

Requirements should state *what* the system should do, whereas design and code state *how*. In practice, however, while the line between the two is not always clear, even in traditional development approaches, it is significantly blurred in MBD. For example, a Simulink model is often considered both the requirements specification and the detailed design specification. Graphical models are often used to help understand requirements. They may also provide a convenient means for facilitating communication between domain experts and software developers. However, Simulink models are *not* requirements. Simulink models contain too many design (implementation) details, making it difficult to see the black-box behavior of a system. Furthermore, a Simulink model lacks a means for specifying nonfunctional requirements and properties of the system (e.g., confidentiality).

3.3.4 What Is Wrong With Requirements Specifications Today?

Many organizations using MBD recognize the importance of separating requirements specification from design. However, the requirements are often written using natural language, and are therefore bound to be ambiguous. Furthermore, the requirements are often incomplete, that is, they specify the required functionality of the system for particular combinations of inputs, but often fail to specify the functionality for all the combinations.

We have also often seen inconsistent requirements specifications, that is, those containing contradictory statements. Using a language with precise syntax and semantics (meaning) helps alleviate these issues. Consider, for example, the requirement captured in the tabular expression [18] shown as Table 3.1. Tabular expressions are one of many ways to specify requirements. However, they offer precise and concise semantics, and are used in the nuclear and aerospace industries due to their understandability. They can be interpreted straightforwardly as if-then-else statements. Consider writing a requirement for driver request arbitration from the Park position in the illustrative example described in Section 3.2.3 that states: "If there is no fault and the component is unlocked, grant the driver's request; otherwise, stay in the current gear." This requirement can be compactly specified as a tabular expression for the Park position as shown in Table 3.1, where each row represents a subexpression of the function such that if a Condition is evaluated to be true, the corresponding Result cell value is the returned output.

Given the requirement specified in Table 3.1, it is straightforward through the use of tool support [19] to verify that

Table 3.1 Requirement for Driver Request Arbitration From Park

fArbRequestFromPark(eDrvrRequest:enum, bUnlocked, bFaulty:bool): enum =

Condition		Result
		eArbRequest
bFaulty		*cPark*
¬bFaulty	*bUnlocked*	*eDrvrRequest*
	¬bUnlocked	*cPark*

the requirement is complete (requiring consideration of all possible inputs) and consistent (ensuring determinism through nonoverlapping input cases), both of which are integral to safety-critical systems, as they raise the confidence in correct system performance in all conditions, and also aid in detecting gaps for the input cases considered.

3.3.5 Who Writes the Requirements Specification?

Ideally, domain experts would write the requirements specification themselves, without the help of software engineers. However, this is seldom the case, with software engineers producing the requirements specification based on communication with domain experts. The knowledge of the domain experts is instrumental to the specification of requirements, but the developer possesses the knowledge of how to specify the requirement precisely and succinctly. While getting requirements right necessitates continual interaction between domain experts and software engineers, there is commonly a disconnect, as they often do not "speak the same language." Specifying requirements such that they are understandable to domain experts, and the use of notations like the aforementioned tabular expressions are integral to the development of a quality requirements specification. MBD notations like Simulink/Stateflow have proven to be useful in this regard, given that they are readable by both domain experts and software engineers.

3.3.6 What Information Should an SRS Contain?

The structure and content of a SRS have been thoroughly investigated, with several standards and templates available [20]. At minimum, an SRS typically consists of the following elements:

Purpose: A clear statement of the system's fundamental reason for existence. This is meant to provide a rudimentary understanding of the system and why it is needed.

Scope: Includes a brief overview of the system to be developed and should indicate the goals and benefits of building the system. It also specifies the boundaries within which these goals are met. An accurate scope definition is important since it is often used by project managers to determine timing and budget estimates.

Functional Requirements: A *functional requirement* specifies an action or feature that needs to be included in the software system in order for the system to be fit for purpose. Table 3.1 is an example of a functional requirement.

Nonfunctional Requirements: A *nonfunctional requirement* specifies a property or quality that the software system shall possess in order to judge its operation. Nonfunctional requirements often specify the performance, security, and usability requirements of the software system, among others.

An SRS should also contain specifications of the tolerances on *accuracies* of outputs, *rationale* justifying the reason for the existence of requirements (with alternatives considered, if any), specifications of *interfaces* documenting how the software communicates with its environment, and documentation of *anticipated changes* to existing requirements so that they may be better accommodated by the eventual design.

Once a preliminary set of requirements can be agreed upon by the domain experts and other stakeholders, and there is a general understanding of *what* the system must do, thought can start being put into *how* the system is going to do what it does. It should be noted that requirements specification is an iterative process that continues in subsequent phases.

3.4 Design: How Will Your Software Do What It Does?

Designing software is similar to design activities in other engineering fields. It is the process of determining how a system will perform its intended functions. The software design process is regularly comprised of two stages: *architectural design* and *detailed software design*. The design starts with determining the software architecture, which is the description of the high-level decomposition of the system into its main components, their interfaces, and interactions between the components. Software architecture is then gradually refined into a detailed design of modules and algorithms. In MBD, the software design refers to the modeling of the software in a language such as Simulink/Stateflow, with the models effectively serving as blueprints for the software implementation, done via automatic code generation.

3.4.1 How Is Design Different From Requirements?

Design is directly driven by the requirements gathered in the previous phase. Models are created and continually modified until a design has been achieved that meets all the requirements. Although closely tied together, it is important to emphasize again that requirements are *not* the same as design models. As previously mentioned, this is one of the most prevalent misconceptions when it comes to MBD, with MathWorks also perpetuating this idea in the recent past [21]. Requirements and design must be viewed as separate entities, and we can illustrate exactly why using the automotive example given in Section 3.2.3.

Table 3.1 specifies a requirement, while Tables 3.2 and 3.3 provide two detailed Stateflow designs which both satisfy this requirement. These Stateflow truth table designs are structured in two sections, where the top subtable defines conditions to check. Should the conditions be evaluated to the values given in the columns, (T, F, or -, representing true, false, or "don't care," respectively), the corresponding action for the column is executed. Actions are defined in the bottom subtable. It is apparent that pinpointing the requirement within these designs is difficult due to the additional design details also included. Moreover, this example demonstrates that multiple, yet distinct, designs can implement the same requirement in different ways. For these reasons, it is imperative to document requirements separately from design. Just as in engineering in general, the motivation for choosing one design over another will lie in the

Table 3.2 First Design Stateflow Truth Table

fArbRequestFromPark(eDrvrRequest:enum, bUnlocked, bFaulty:bool): enum =

#	Condition	1	2	3	4	5	6	7	8	9	10	11
1	*eDrvrRequest == cPark*	T	F	F	F	F	F	F	F	F	F	-
2	*eDrvrRequest == cReverse*	F	T	F	F	T	F	F	T	F	F	-
3	*eDrvrRequest == cNeutral*	F	F	T	F	F	T	F	F	T	F	-
4	*eDrvrRequest == cDrive*	F	F	F	T	F	F	T	F	F	T	-
5	*bUnlocked*	-	T	T	T	-	-	-	-	-	-	-
6	*bFaulty*	-	F	F	F	T	T	T	-	-	-	-
	Actions	1	2	3	4	1	1	1	1	1	1	1

#	Action
1	*eArbRequest = cPark*
2	*eArbRequest = cReverse*
3	*eArbRequest = cNeutral*
4	*eArbRequest = cDrive*

Table 3.3 Second Design Stateflow Truth Table

fArbRequestFromPark(eDrvrRequest:enum, bUnlocked, bFaulty:bool): enum =

#	Condition	1	2	3
1	*bFaulty*	T	F	F
2	*bUnlocked*	-	T	F
	Actions	1	2	1

#	Action
1	*eArbRequest = cPark*
2	*eArbRequest = eDrvrRequest*

added need to satisfy other requirements or accommodate constraints. For example, if the component containing design implementing the requirement from Table 3.1 has a tight timing requirement, the second design may be used due to its more efficient condition checking. However, if maintainability over different, but similar, software versions containing this component, is the bigger concern, the first design will more likely be used, as will be explained later in this section.

3.4.2 What Are Important Principles of Software Design?

It is well known in software engineering that good designs lead to high-quality software systems. For systems other than trivial examples, it is necessary to *decompose*, or break up, the system into manageable modules in order to improve its reusability, overcome complexity, and to divide labor. There are typically several ways of decomposing a system. The criteria used in the decomposition of a system plays a significant role in determining the quality of a design. One of the most important principles in software design is *design for change* [22] which prescribes that a developer needs to be able to anticipate changes that the system might undergo, and design software capable of accommodating those changes. For example, when designing powertrain software, engineers need to *anticipate* powertrain configurations that might have to be supported in the future, and design software so that, if the change is made, the effect of the change will be localized as much as possible. Closely related to the *design for change* and *anticipation of change* principles is the concept of a *software product line*. A product line necessitates a core architecture of common functionality across the various configurations, but will also provide the ability to include variations in order to create different products within the line. For example, a large part of electrified powertrain software can be reused throughout different powertrain configurations. All of the software versions corresponding to different powertrain configurations will constitute products within a software product line. As another example, the model shown in Table 3.2 was developed to satisfy the requirement from Table 3.1, but was also devised with the product line approach in mind, because the logic it implements varies only slightly with different vehicle variants. More precisely, while the conditions listed in the columns of the first table of Table 3.2 remain the same for each product in the product line, the set of actions on these conditions is the only part of the design that varies throughout the different products within the product line. Roughly speaking, the actions are encoded as calibrations, so that they are easy to change, and maintain. Calibrations, in fact, are often used to implement variability in software across products within a software product line.

The mechanism crucial in implementing *design for change* in software engineering is *information hiding* [22]. Information hiding seeks to decompose a system such that modules each "hide" a requirement or design decision that is likely to change, that is, the interface of the module does not reveal its inner workings. Typically, design decisions creep into the interfaces of the modules, making them context-dependent, and not easily modifiable or reusable. Design decisions typically correspond to hardware, behavior, and software design decisions which are likely to change in the future, and hiding their details within a module will make future changes easier to accommodate. Continuing with the aforementioned electrified powertrain software example, a module that will "hide" the powertrain architecture from the rest of powertrain software represents a *hardware hiding module*. However, while the principle of information hiding has fared well in traditional software development paradigms, it might not be as useful and widely applicable in MBD. We are currently undertaking research into the role of information hiding in MBD.

3.4.3 How Does Simulink Support the Application of Software Engineering Principles?

For MBD, Simulink enables the introduction of various levels of hierarchy in order to decompose a system into various levels of abstraction. Unfortunately, a challenge in Simulink is understanding how to employ information hiding, how designs will benefit from it, as well as how to decompose a system into reusable modules. The *subsystem* is the accepted Simulink equivalent of a module, however, they are neither reusable, nor do they effectively encapsulate their internal design. Degrees of reusability can be achieved with other mechanisms such as libraries, model references, function-call subsystems, code reuse subsystems, and Simulink functions, however, they all fail to encapsulate their internals with respect to hidden data flow [23]. For example, Data Store Memory blocks are able to bypass the typical inport/outport interface of a subsystem, and read/write data directly from/in the subsystem. Adding explicit interfaces which include Data Store Memory blocks such as those described in [23] can alleviate this problem. However, a new block mechanism within the Simulink language is needed; one which restricts hidden data flow to effectively encapsulate data, as well as be easily reused in multiple locations of a model. Such a mechanism is not currently available and presents itself as a challenge when employing information hiding in Simulink designs. Research into the development of such mechanisms is needed.

Furthermore, Simulink lacks self-documenting capabilities of imperative programing languages. For instance, an analog of a module interface in C, as defined in C header files, does not exist in Simulink [23].

3.4.4 How Can Guidelines Help?

When it comes to achieving a good design, as with most languages, there are conventions and guidelines available which give best practices that should be adhered to. Likewise, for Simulink/Stateflow, standards such as [24,25] have been developed with the aim of facilitating desirable model qualities, mostly readability. Making models readable with appropriate block colors and positions is comparable to including white spaces and new lines in textual languages, and makes a difference when it comes to achieving qualities such as modifiability and maintainability.

Nevertheless, in working with industrial-sized models from OEMs and the currently available guidelines, we have noticed shortcomings in the guidelines in addressing actual design principles, such as modularity. For example, using global variables in traditional programming languages is strongly regarded as bad practice because global variables hinder encapsulation, reuse, and understandability. However, modeling guidelines for Simulink typically do not recommend against the use of analogous constructs such as Data Store Memory blocks at the top-level of models (which would be analogous to them being declared as global variables), or above their needed scope. Such a recommendation can easily be formulated and automated, as done in Ref. [26] with the *Data Store Push-Down Tool*. In general, more guidelines and supporting tools are needed, which aim to increase the use of other important software engineering principles.

3.4.5 What Information Should a Software Design Document Contain?

As with other traditional development approaches, designs in MBD must be properly documented. A software design description (SDD) is an artefact documenting the design of the software system and describing how the system will be structured in order to satisfy its requirements. An SDD effectively translates the requirements from the SRS into a representation using software components, interfaces, and data. Commonly used templates which outline the content and format of compiling an SDD exist [27]. At minimum, an SDD typically consists of the following elements:

Purpose: A clear statement describing what the system is ultimately meant to accomplish. It is meant to reinforce the understanding of why the system needs to be developed.

Rationale: Provides justification for the chosen design. This often includes a description and justification of the design decisions that were made in the development of a module, and a list of the alternatives that were considered, along with reasons why they were rejected.

Interface Design: Describes the intended behavior of a module from an external viewpoint, such that other entities can interact with the module without knowing its internal design. This should include the any imported modules, inputs, outputs, and their types, ranges, etc.

Internal Design: Describes the internal structure of a module, including subsystems, algorithms, internal variables/data, and constants.

Anticipated Changes: A list of the ways in which a module is expected to change in the future. This offers insight into the future direction of the development of a module. In this way, one can *design for change* so that when requirements of the system change, the design can accommodate those changes with only moderate modifications, rather than with complete overhauls.

Although the need for documenting Simulink models has been recognized in industry, to the best of our knowledge, there has not been any research on how this is to be done. Our own efforts show that the principles and content of an SDD from traditional software engineering equally apply to documentation of Simulink models, and we have been working to develop a template for an SDD for Simulink models.

3.4.6 Are Models Documentation?

In MBD, we are often met with the "models are documentation" fallacy that we believe has further perpetuated the lack of proper documentation across industries using MBD. However, any engineer responsible for maintaining real-world industrial-size Simulink models understands that a Simulink model is notoriously hard to reverse engineer or maintain without additional information about the model that can be documented in an SDD. For example, Simulink lacks facilities to explicitly represent the interface inputs/outputs of a model/subsystem. This issue was discussed and suggestions were made in [23]. Also, a model does not contain rationale for design decisions. However, experience teaches us that documenting rationale is crucial for proper software development and maintenance.

We illustrate the importance of having a good SDD by an anecdotal story from our collaboration with one of our industrial partners. Their newly hired engineer was tasked with maintaining a Simulink model implementing algorithms within his expertise area. There was no documentation associated with the model. Although the engineer was very familiar with the model's algorithms and their application, comprehending the model took approximately 2 months due to the fact that no requirements specification, and particularly, design documentation, existed for this model. As a result, every part of the model had to be manually examined and understood. After reverse engineering the model, the engineer asked for our help with documenting the model to significantly ease the maintenance efforts in the future. This is not the only instance of such setbacks we saw, and it clearly illustrates that even a domain expert, with all of the relevant background knowledge, is still hampered significantly by a lack of documentation. Again, this is a clear example that the Simulink model is *not* the requirements, nor effective documentation in and of itself.

3.4.7 What Is Wrong With Software Design Documentation Today?

In general, it is a common attitude that SDDs are ultimately nonessential to the deployment of embedded software. The companies that develop and maintain large and complex embedded software in Simulink, also develop and maintain a large number of SDDs documenting the designs. For example, a company we worked with documents every software feature (i.e., a large Simulink model) with an SDD. To improve the documentation, the company developed a template defining the format and content of SDDs and then distributed it to developers in charge of models' maintenance. However, the template very loosely defined the content of SDDs, partly due to the use of undefined terminology. This resulted in developers subjectively interpreting the template, leading to inconsistent documentation throughout different features of the same software. The SDDs are also consequently ambiguous and incomplete. Under-defined content of documentation is a general (not only SDD) software documentation problem, ultimately rendering the resulting documentation meaningless. Instead, the template for documentation should define the structure of the documentation, using well-defined terminology that includes explanation of all relevant terms, as well as the instructions for the developers on the required content. Improving documentation is not a short-term project—consequently, the managers consider it a burden on the development/maintenance process already under tight resource constraints. We feel, however, that the benefits of producing and maintaining proper documentation would by far outweigh its costs.

Additionally, a challenge we have encountered in industry, especially those with fast development cycles, is that SDDs are not always kept up-to-date. We contend that every model change should also necessitate a change in the associated SDD. Ideally, the change management should be built into software development environments with revision control, with rules requiring that changes to models are not allowed without an updated SDD.

3.5 Implementation: Generating Code

3.5.1 Why Is Code Generation Crucial to the Success of MBD?

Automatic code generation in an MBD process is vital to the cost effectiveness of development. It eliminates the manual effort in coding from design, therefore, accelerating the process while decreasing the chance of errors when compared to manual coding from requirements or models. For example, GM has attributed the success of the Chevrolet Volt's development to automatic code generation [28]. Since code is automatically generated from design, traceability links are also automatically generated. Tools exist that automatically generate code from Simulink models and have been widely used in the industry (e.g., MathWorks Embedded Coder, dSPACE TargetLink). Any manual modification of the code after code generation is strongly not recommended, given the high chance of introducing errors, and maintainability issues—the manual modifications will be overwritten upon code regeneration.

While verification that the code implements the Simulink design is still needed (performed by, e.g., *back-to-back testing*[2] that is well supported by current tools), verification efforts can typically be reduced by using the "proven in use" argument behind commercial code generation tools—the fact that those tools have extensively been successfully used in different applications for a reasonable amount of time. Some industries go further by certifying code generators, additionally reducing the effort needed for verification of code against design.

Automatic code generation enables a variety of applications including SiL, PiL, HiL, and rapid prototyping. It allows for quick generation of code from Simulink controller implementations for deployment code on a desktop machine, instruction set simulators, or target (the microprocessor). Further more, for HiL, e.g., the plant model can also be coded into C (whether from Simulink or another physical modeling tool more appropriate for plant modeling) and used in real time. The embedded code generated for ECUs should also run in real time, satisfy efficiency requirements (speed, memory usage), integration with legacy code requirements, etc.

3.5.2 What Are the Limitations of Code Generation?

Not all of the Matlab language and Simulink constructs are supported by code generation tools. Furthermore, while efficiency of model-generated code is comparable to hand code[3], the efficiency of code can typically be increased by hand coding

[2]Back-to-back testing checks whether the outputs of the model and code are the same for the same inputs.
[3]In fact, model-generated code can outperform handwritten code [29].