Deterministic Reactive Programming for Cyber-physical Systems

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden,
Fakultät Informatik,

eingereicht von

Christian Menard

Gutachter:

Prof. Jeronimo Castrillon
Technische Universität Dresden

Prof. Stephen Edwards
Columbia University

Tag der Verteidigung:

25.04.2024

Deterministic Reactive Programming for Cyber-physical Systems

Christian Menard

January 5, 2024

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class. The document was processed by LuaTeX, and the bibliography was processed by BibLaTeX and Biber. The open-source Libertinus family of fonts is used for all text.

To my grandfather,
who always encouraged my curiosity.

Abstract

Today, cyber-physical systems (CPSs) are ubiquitous. Whether it is robotics, electric vehicles, the smart home, autonomous driving, or smart prosthetics, CPSs shape our day-to-day lives. Yet, designing and programming CPSs becomes evermore challenging as the overall complexity of systems increases. CPSs need to interface (potentially distributed) computation with concurrent processes in the physical world while fulfilling strict safety requirements. Modern and popular frameworks for designing CPS applications, such as ROS and AUTOSAR, address the complexity challenges by emphasizing scalability and reactivity. This, however, comes at the cost of compromising determinism and the time predictability of applications, which ultimately compromises safety. This thesis argues that this compromise is not a necessity and demonstrates that scalability can be achieved while ensuring a predictable execution.

At the core of this thesis is the novel reactor model of computation (MoC) that promises to provide timed semantics, reactivity, scalability, and determinism. A comprehensive study of related models indicates that there is indeed no other MoC that provides similar properties. The main contribution of this thesis is the introduction of a complete set of tools that make the reactor model accessible for CPS design and a demonstration of their ability to facilitate the development of scalable deterministic software.

After introducing the reactor model, we discuss its key principles and utility through an adaptation of reactors in the Discrete Events for AUTOSAR (DEAR) framework. This framework integrates reactors with a popular runtime for adaptive automotive applications developed by AUTOSAR. An existing AUTOSAR demonstrator application serves as a case study that exposes the problem of nondeterminism in modern CPS frameworks. We show that the reactor model and its implementation in the DEAR framework are applicable for achieving determinism in industrial use cases.

Building on the reactor model, we introduce the polyglot coordination language Lingua Franca (LF), which enables the definition of reactor programs independent of a concrete target programming language. Based on the DEAR framework, we develop a full-fledged C++ reactor runtime and a code generation backend for LF. Various use cases studied throughout the thesis illustrate the general applicability of reactors and LF to CPS design, and a comprehensive performance evaluation using an optimized version of the C++ reactor runtime demonstrates the scalability of LF programs. We also discuss some limitations of the current scheduling mechanisms and show how they can be overcome by partitioning programs.

Finally, we consider design space exploration (DSE) techniques to further improve the scalability of LF programs and manage hardware complexity by automating the process of allocating hardware resources to specific components in the program. This thesis contributes the Mocasin framework, which resembles a modular platform for prototyping and researching DSE flows. While a concrete integration with LF remains for future work, Mocasin provides a foundation for exploring DSE in Lingua Franca.

"Let's think the unthinkable, let's do the undoable. Let us prepare to grapple with the ineffable itself, and see if we may not eff it after all."

— Douglas Adams

Acknowledgements

"We need to understand that if we all work on inclusion together, it's going to be faster, broader, better, and more thorough than anything we can do on our own."

— Ellen Pao

First and foremost, I owe my gratitude to my advisor, Jeronimo Castrillon. Shortly after he became a professor at TU Dresden and finally filled the vacant Chair of Compiler Construction with life again, he took me on as a student to work on my diploma thesis. From these first days on, he has given me complete freedom and trust while pushing me to set and pursue higher goals. Over the years, this environment has transformed me into an independent, confident researcher who is ready to tackle new challenges on the path ahead. I would not be writing this thesis if not for him.

I also owe my gratitude to Andrés Goens in a similar capacity. He was my advisor while I was working on my diploma thesis and has continued to be both a mentor and friend since then. It was Andrés who brought the reactor model to my attention and introduced me to Marten Lohstroh and Edward A. Lee. Likely unbeknownst to the both of us, this planted the seed for a fruitful collaboration and for the core contributions of this thesis.

I also would like to thank all my current and former coworkers at the Chair for Compiler Construction. This includes Justus Adam, Hasna Bouraoui, Alexander Brauckmann, Sebastian Ertel, Andrés Goens, Fazal Hameed, Gerald Hempel, Hamid Farzaneh, Clément Fournier, Karl Friebel, Sven Karol, Asif Khan, Robert Khasanov, Nesrine Khouzami, Steffen Köhler, Galina Kozyreva, João Paulo Cardoso de Lima, Norman Rink, Julian Robledo, Lars Schütze, and Felix Suchert. Each and every one of you has made the office a lively and diverse space that I will miss dearly. A special thanks goes to Conny Okuma who tirelessly supports the team and always keeps on top of all our small and big requests.

I am also grateful to all the students that I had the pleasure to work with. This includes Maiko Brants, Hugo Forrat, Clément Fournier, Johannes Hayeß, Hannes Klein, Anton Landgraf, Lucian McIntyre, Marcus Rossel, Robert Scheffel, Christoph Schröter, Franziska Schwenke, Tassilo Tanneberger, and Felix Teweleitt. Not only did you help me realize small and big side projects, but you have also been a constant source of inspiration. I have learned a lot from guiding you, and it has been a pleasure to see you grow and develop. I am glad to see that many of you became successful engineers and researchers; some of you even became coworkers. Finally, I would like to thank specially Clément Fournier, Marcus Rossel, and Tassilo Tanneberger for their exceptional contributions. Keep up the brilliant work!

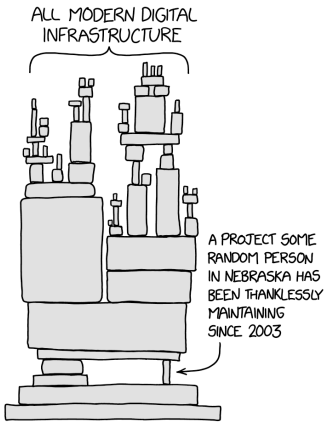
The last years of my PhD have been strongly influenced by the Lingua Franca community. In particular, the collaboration with Marten Lohstroh and Edward A. Lee at UC Berkeley had a strong impact, both on this thesis and on me personally. It is not an overstatement to say that this collaboration has been life-changing. I am truly grateful for meeting both of you and for being able to continue working with you beyond my PhD. You have welcomed me warmly, openly shared your ideas, invited me to contribute my own ideas, and treated me as an equal right from the beginning, despite my greenness. I appreciate your trust in me, and I am very much looking forward to finding out what else we will achieve together.

I would also like to thank everyone else in the Lingua Franca community for their contributions to the project, for their eager spirits and ideas, for the many fruitful discussions we had, and for the friendly and inspirational environment that they create. In particular, I would like to mention Soroush Bateni, Sören Domrös, Peter Donovan, Reinhard von Hanxleden, Hokeun

Kim Shaokai Lin, and Alexander-Schulz Rosengarten, but there are many more who have impacted me and this thesis in one way or another. I appreciate all of you.

I also want to thank all of my other coauthors that I have not mentioned personally so far. There are too many to list them all. However, I would like to specially mention Matthias Jung, who openly shared his earlier work with me when we realized that we were working towards the same goal from two different directions, and who suggested work on a joint paper. Also, I want to specially thank Jason-Lowe Power for all of his efforts in the gem5 project and for inviting me to contribute to the new gem5 paper.

The work conducted in this thesis would not have been possible without funding. In particular, this thesis was supported by the center for advancing electronics Dresden (cfaed), by the German Research Foundation (DFG), and the German Federal Ministry of Education and Research (BF). I would like to specially thank the Software Campus for accepting my application, for supporting me with outstanding workshops, and for providing me with funding for my own projects.



Source: xkcd, licensed under CC BY-NC 2.5

I would also like to use this opportunity to express my appreciation and gratitude to the entire open-source community. Contributions to open-source projects ensure that our digital infrastructure keeps running and are a key enabler for the research conducted in this thesis, but more importantly, for all the brilliant research conducted across all disciplines of science. Whether you have a prominent name like Linus Torvalds or you are the random person in Nebraska maintaining an unknown project, thank you! I hope you accept the research artifacts that I created while working on this thesis as my humble contribution to the overall effort.

Further, I would like to thank all the people who dedicate their lives to music. While I never understood how to play an instrument myself, music plays an important role in my life. Listening to great music and visiting concerts presents an opportunity for breaking out of the day-to-day work and is a source of energy and inspiration for me. I would like to specially thank the creator of the "5 Hours of Relaxing Psychedelic Space Rock" playlists and all the amazing musicians whose music is featured in them. Besides maté, it was these playlists that kept me going while writing this thesis.

Finally, but most importantly, I want to thank my friends and family. Many of my friends have accompanied and supported me for more than a decade (some even for two). Your friendship is invaluable to me. I thank you for giving me a family right when I needed one, for accepting me for who I was, and for helping me become who I am today. I have learned a lot from you, and I appreciate all the time that we spend together. I am also grateful for my brother, whom I am delighted to have in my life. Above all, I thank my partner, Nicole. Thank you for believing in me, for supporting me, for challenging me to grow, for giving me the freedom to pursue my goals, and for reminding me of all the other beautiful aspects of life. Lastly, I am grateful for Nicole's family, who has openly and warmly welcomed and supported me as one of their own.

Publications

Several ideas, figures and arguments that are presented in this thesis have been published in prior work. The following lists all the publications that I have co-authored and that are cited in this thesis in reverse chronological order:

- Marten Lohstroh, Soroush Bateni, Christian Menard, Alexander Schulz-Rosengarten, Jeronimo Castrillon, and Edward A. Lee (2023). *Deterministic Coordination Across Multiple Timelines*. In: *ACM Transactions on Embedded Computing Systems*. ISSN: 1539-9087
- Edward A. Lee, Ravi Akella, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard (2023). *Consistency Vs. Availability in Distributed Cyber-Physical Systems*. In: *ACM Transactions on Embedded Computing Systems* 22.5s. ISSN: 1539-9087
- Soroush Bateni, Marten Lohstroh, Hou Seng Wong, Hokeun Kim, Shaokai Lin, Christian Menard, and Edward A. Lee (Sept. 2023). *Risk and Mitigation of Nondeterminism in Distributed Cyber-Physical Systems*. In: *21st ACM-IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE)*, pp. 1–11
- Christian Menard, Marten Lohstroh, Soroush Bateni, Matthew Chorlian, Arthur Deng, Peter Donovan, Clément Fournier, Shaokai Lin, Felix Suchert, Tassilo Tanneberger, Hokeun Kim, Jeronimo Castrillon, and Edward A. Lee (2023). *High-Performance Deterministic Concurrency Using Lingua Franca*. In: *ACM Transactions on Architecture and Code Optimization*. Just Accepted. ISSN: 1544-3566
- Jeronimo Castrillon, Karol Desnos, Andrés Goens, and Christian Menard (Jan. 2023). *Dataflow Models of Computation for Programming Heterogeneous Multicores*. In: *Handbook of Computer Architecture*. Ed. by Anupam Chattopadhyay et al. Singapore: Springer Nature Singapore. ISBN: 978-981-15-6401-7
- Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard (2023). *Trading Off Consistency and Availability in Tiered Heterogeneous Distributed Systems*. In: *Intelligent Computing* 2
- Reinhard von Hanxleden, Edward A. Lee, Hauke Fuhrmann, Alexander Schulz-Rosengarten, Sören Domrös, Marten Lohstroh, Soroush Bateni, and Christian Menard (2022). *Pragmatics Twelve Years Later: A Report on Lingua Franca*. In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*. Springer Nature Switzerland, pp. 60–89
- Robert Khasanov, Julian Robledo, Christian Menard, Andrés Goens, and Jeronimo Castrillon (Sept. 2021). *Domain-Specific Hybrid Mapping for Energy-Efficient Baseband Processing in Wireless Networks*. In: *ACM Transactions on Embedded Computing Systems (TECS). Special issue of the International Conference on Compilers, Architecture, and Synthesis of Embedded Systems (CASES)* 20.5s. ISSN: 1539-9087
- Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee (2021). *Toward a Lingua Franca for Deterministic Concurrent Systems*. In: *ACM Transactions on Embedded Computing Systems* 20.4, pp. 1–27
- Christian Menard, Andrés Goens, Gerald Hempel, Robert Khasanov, Julian Robledo, Felix Teweleitt, and Jeronimo Castrillon (Jan. 2021). *Mocasin—Rapid Prototyping of Rapid Prototyping Tools: A Framework for Exploring New Approaches in Mapping Software to Heterogeneous Multicores*. In: *Proceedings of the 2021 Drone Systems Engineering and Rapid*

"Science knows no country, because knowledge belongs to humanity, and is the torch which illuminates the world. Science is the highest personification of the nation because that nation will remain the first which carries the furthest the works of thought and intelligence."

— Louis Pasteur

Simulation and Performance Evaluation: Methods and Tools, co-located with 16th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC). DroneSE and RAPIDO '21. Budapest, Hungary: Association for Computing Machinery, pp. 66–73. ISBN: 9781450389525

- Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard (2021). *Quantifying and Generalizing the CAP Theorem*
- Robert Wittig, Andrés Goens, Christian Menard, Emil Matus, Gerhard P. Fettweis, and Jeronimo Castrillon (Oct. 2020). *Modem Design in the Era of 5G and Beyond: The Need for a Formal Approach*. In: *Proceedings of the 27th International Conference on Telecommunications (ICT)*. Virtual. Bali, Indonesia, pp. 1–5
- Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A. Lee (Sept. 2020). *A Language for Deterministic Coordination Across Multiple Timelines*. In: *2020 Forum for Specification and Design Languages (FDL)*. Kiel, Germany, pp. 1–8
- Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian (July 2020). *The Gem5 Simulator: Version 20.0+*. In: *arXiv preprint arXiv:2007.03152*
- Christian Menard, Andrés Goens, Marten Lohstroh, and Jeronimo Castrillon (Mar. 2020). *Achieving Determinism in Adaptive AUTOSAR*. In: *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE)*. DATE '20. Grenoble, France: IEEE, pp. 822–827. ISBN: 978-3-9819263-4-7
- Andrés Goens, Christian Menard, and Jeronimo Castrillon (Sept. 2018). *On the Representation of Mappings to Multicores*. In: *Proceedings of the IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-18)*. Vietnam National University, Hanoi, Vietnam, pp. 184–191. ISBN: 978-1-5386-6689-0
- Jeronimo Castrillon, Matthias Lieber, Sascha Klüppelholz, Marcus Völp, Nils Asmussen, Uwe Assmann, Franz Baader, Christel Baier, Gerhard Fettweis, Jochen Fröhlich, Andrés Goens, Sebastian Haas, Dirk Habich, Hermann Härtig, Mattis Hasler, Immo Huismann, Tomas Karnagel, Sven Karol, Akash Kumar, Wolfgang Lehner, Linda Leuschner, Siqi Ling, Steffen Märcker, Christian Menard, Johannes Mey, Wolfgang Nagel, Benedikt Nöthen, Rafael Peñaloza, Michael Raitza, Jörg Stiller, Annett Ungethüm, Axel Voigt, and Sascha Wunderlich (July 2018). *A*

Hardware/software Stack for Heterogeneous Systems. In: *IEEE Transactions on Multi-Scale Computing Systems* 4.3, pp. 243–259. ISSN: 2332-7766

- Christian Menard, Matthias Jung, Jeronimo Castrillon, and Norbert Wehn (July 2017). *System Simulation with gem5 and SystemC: The Keystone for Full Interoperability*. In: *Proceedings of the IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS)*. IEEE. Pythagorion, Greece, pp. 62–69. ISBN: 978-1-5386-3437-0
- Christian Menard, Andrés Goens, and Jeronimo Castrillon (Nov. 2016). *High-Level NoC Model for MPSoC Compilers*. In: *Proceedings of the IEEE Nordic Circuits and Systems Conference (NORCAS'16)*. NORCAS. Copenhagen, Denmark

The following publications are co-authored by me but not cited in this thesis:

- Andrés Goens, Christian Menard, and Jeronimo Castrillon (July 2019). *On Compact Mappings for Multicore Systems*. In: *Proceedings of the IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS)*. Ed. by D. Pnevmatikatos, M. Pelcat, and M. Jung. Vol. 11733. IEEE. Pythagorion, Greece: Springer, Cham, pp. 325–335. ISBN: 978-3-030-27561-7
- Fazal Hameed, Christian Menard, and Jeronimo Castrillon (Oct. 2017). *Efficient STT-RAM Last-Level-Cache Architecture to replace DRAM Cache*. In: *Proceedings of the International Symposium on Memory Systems (MemSys'17)*. MEMSYS '17. Alexandria, Virginia: ACM, pp. 141–151. ISBN: 978-1-4503-5335-9

Contents

1 Introduction

Today, cyber-physical systems (CPSs) are ubiquitous. Whether it is robotics, electric vehicles, the smart home, autonomous driving, or smart prosthetics, CPSs shape our day-to-day lives. Also, several major technology trends have CPS at their core. CPS is nothing less than the enabler of the Fourth Industrial Revolution.[1] In the energy sector, CPSs are essential to support the ongoing decarbonization, as it requires transforming the current centralized power grids, with only a view of major power plants, into a decentralized network. The envisioned *smart grid* is a CPS that intelligently and reliably coordinates power consumption and production across all stakeholders.[2] Finally, visions like the *smart city* rely on CPS technology and have the potential to substantially change our societies.[3]

1: Jazdi 2014, *Cyber Physical Systems in the Context of Industry 4.0*; Pivoto et al. 2021, *Cyber-Physical Systems Architectures for Industrial Internet of Things Applications in Industry 4.0: a Literature Review*.

2: Yu and Xue 2016, *Smart Grids: a Cyber-Physical Systems Perspective*; Mazumder et al. 2021, *A Review of Current Research Trends in Power-Electronic Innovations in Cyber-Physical Systems*.

3: Cassandras 2016, *Smart Cities as Cyber-Physical Social Systems*; Trencher 2019, *Towards the Smart City 2.0: Empirical Evidence of Using Smartness As a Tool for Tackling Social Challenges*; Puliafito et al. 2021, *Smart Cities of the Future as Cyber Physical Systems: Challenges and Enabling Technologies*.

There are many definitions for CPSs. CPSs are unique in that they combine computation, the *cyber*, with the ability to sense and influence the environment, the *physical*. E. A. Lee and Seshia provide a simple and intuitive definition:

> The term cyber-physical system (CPS) … refer[s] to the integration of computation with physical processes. In CPS, embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa.[4]

4: E. A. Lee and Seshia 2016, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*.

We will adopt this rather traditional view of CPS in this thesis. In some domains, for instance, in Industry 4.0, the term cyber-physical system often also implies a networked system where multiple robots or entire manufacturing lines are interconnected.

Precisely due to the combination of the cyber and the physical, the design of CPSs remains challenging. E. A. Lee and Seshia note:

> As an intellectual challenge, CPS is about the *intersection*, not the union, of the physical and the cyber. It is not sufficient to separately understand the physical components and the computational components. We must instead understand their interaction.[4]

This is in contradiction to many of the abstractions that are commonly used in computer science. In fact, entire domains in computer science rely on the fact that we can abstract away the physical nature of computation. Rarely, a computer scientist is concerned with the physical reality of electrons moving through silicon when writing software. With virtualization, cloud computing, and serverless computing,[5] we also do not need to be concerned with where the computation is performed physically.

5: Kounev et al. 2023, *Serverless Computing: What It Is, and What It Is Not?*

In CPS, however, *where* and *when* we compute something matters. Designing CPSs requires interfacing computation with the physical world, and thus we need to expose properties of the physical world to the computation. Therefore, there is a need to rethink the fundamental abstractions of computer science to account for the requirements of CPSs. Most importantly, useful abstractions for computation in CPS should include a notion of the passage of time.[6]

6: E. A. Lee 2009, *Computing Needs Time*.

This thesis investigates the novel *reactor model*,[7] which promises to bridge the gap between the cyber and the physical. This includes a full implementation of the reactor model, novel tooling that facilitates the creation of scalable and efficient programs, and an evaluation thereof.

7: Lohstroh, Romeo, et al. 2019, *Reactors: A Deterministic Model for Composable Reactive Systems*; Lohstroh 2020, *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*.

1.1 Models of Computation

"All models are wrong, but some are useful" is a famous aphorism that is attributed to George Box.[8] A model is a mathematical abstraction that captures the predictable characteristics of a system. Box's aphorism acknowledges that no model is perfect, as they are generally too simple to capture the complexity of reality. Yet, some models are useful, as they predict some aspects of the physical world with remarkable accuracy. Models provide a tool that facilitates focusing on specific problems by reducing the scope to a limited set of rules instead of considering reality in its full complexity. A *useful* model allows us to focus on the right set of rules that are required for accurately predicting properties that are relevant for a particular problem.

8: Box 1979, *Robustness in the Strategy of Scientific Model Building*.

Edward A. Lee observes that to assess the usefulness of a model, we need to distinguish between *scientific* models and *engineering* models.[9] While a scientific model is useful when it can accurately emulate a physical system, an engineering model is useful when we can build physical systems that emulate the model. In the CPS context, a scientific model could be, for instance, a simulation of a moving robot. An engineering model, however, provides a set of rules that allows engineers to develop an application that controls the movement of the robot without a full understanding of how this movement is realized physically.

9: E. A. Lee 2018, *Plato and the Nerd—The Creative Partnership of Humans and Technology*.

This view of engineering models translates seamlessly into the *hourglass model* in platform-based design.[10] In the hourglass model, a system is composed of multiple layers. The centerpiece is a model that consists of a limited set of assumptions and rules. Building on top of this model, engineers can develop support layers and a rich variety of applications. On the lower layers, engineers can independently develop a physical realization of the model that is faithful to its assumptions and rules. Thus, the model facilitates a separation of concerns. The engineers developing the architecture do not need to know the specifics of the applications that might be built on top of the stack, and the application engineers do not need to be concerned with the specifics of the architecture layer. Alberto Sangiovanni-Vincentelli has coined this "freedom from choice," as application developers do not need to worry about the details of a physical realization.[11]

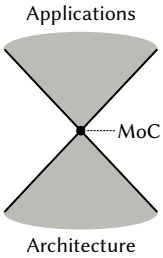


Figure 1.1: The hourglass model.

10: Sangiovanni-Vincentelli 2002, *Defining Platform-Based Design*; Beck 2019, *On the Hourglass Model*.

11: E. A. Lee 2019, *Freedom From Choice and the Power of Models: In Honor of Alberto Sangiovanni-Vincentelli*.

In computer science, a model of computation (MoC) describes some properties of a computation, like *how*, *when*, *where*, or *what* we compute. The first MoCs are attributed to Alan Turing, who introduced his concept of a "computing machine,"[12,13] Alonzo Church, who proposed the λ-calculus,[14] and Stephen Kleene, who introduced the concept of recursive functions.[15] These first MoCs belong to the category of scientific models. They helped find answers to the question "What is computable?", but they are not very helpful when engineering systems.

12: Commonly called the Turing machine.

13: Turing 1937, *On Computable Numbers, With an Application To the Entscheidungsproblem*.

14: Church 1936, *An Unsolvable Problem of Elementary Number Theory*.

15: Kleene 1936, *General Recursive Functions of Natural Numbers.*

MoCs that fall into the category of engineering models, however, restrict computation to a limited but useful set of rules.[16] In the hourglass model, they provide an interface between the upper application layers and the lower architecture layers that facilitate the computation. The rules of the MoC control how an application is executed and also provide a framework for reasoning about some properties of the application. In CPS design, we can leverage well-chosen MoCs to bridge the gap between the cyber and the physical. However, before we can consider MoCs that are useful in this context, we need a better understanding of the requirements and fundamental design challenges in CPS design.

16: The computer science community also uses the term *computing paradigm*, which is synonymous.

1.2 Challenges and Requirements for CPS Design

Interfacing the cyber with the physical brings unique challenges for CPS design. These challenges have been discussed thoroughly in the literature.[17] This section provides an overview of the main challenges that are relevant in the context of this thesis.

17: Stankovic et al. 2005, *Opportunities and Obligations for Physical Computing Systems*; Henzinger and Sifakis 2006, *The Embedded Systems Design Challenge*; E. A. Lee 2008, *Cyber Physical Systems: Design Challenges*; Kopetz 2011, *Real-Time Systems: Design Principles for Distributed Embedded Applications*; Derler, E. A. Lee, and Sangiovanni Vincentelli 2012, *Modeling Cyber-Physical Systems*; Seshia et al. 2017, *Design Automation of Cyber-Physical Systems: Challenges, Advances, and Opportunities*; Marwedel 2021, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*.

1.2.1 Concurrency

CPSs are inherently concurrent. Processes performing computation overlap in time with physical processes in the environment. Thus, a MoC for CPS design requires at least a mechanism for concurrent composition of the computation with physical processes.[18] However, concurrency is useful beyond modeling interactions with the physical world. Most often, CPS applications are naturally structured concurrently. Commonly, there are multiple computation processes that are concerned with different aspects of the system and that may execute, to some extent, independently.

18: E. A. Lee 2008, *Cyber Physical Systems: Design Challenges*.

Also, CPSs are becoming increasingly distributed. In a distributed system, computation is not performed on a single computer, but multiple computations execute concurrently on a set of networked computers, which coordinate the execution by exchanging network messages. A modern car, for instance, has between 30 and 150 electronic control units (ECUs). In a recent report, Volvo disclosed that their cars have more than 120 ECUs that are connected via more than 7,000 signals.[19] A useful MoC in this context should allow for expressing concurrent computation across a multitude of networked nodes, but it should also allow for structuring the program vertically and horizontally in a plausible way and to isolate independent components, allowing programmers and engineers to focus on subsets of the program that are only relevant in a specific context.

19: Antinyan 2020, *Revealing the Complexity of Automotive Software*.

1.2.2 Safety, Reliability, Testability and Determinism

Since CPSs directly interface with the physical environment, the actions they take may have an immediate impact on the environment. Depending on the physical capabilities of the system, this impact may include injuries affecting the health of humans and animals or damage to property and environmental structures. A *safe* system does not expose its surroundings to the risk of harm. A major factor in assessing the overall safety of a system is *functional safety*, which requires that the system operates correctly in response to its inputs and reacts predictably in the event of failure.

While a safe system is free of accidents, a *reliable* system accomplishes the specified tasks and is free of failures.[20] A CPS may fail not only due to a physical component breaking but also due to problematic behavior in the software. Both achieving functional safety and high reliability require thorough testing of the involved software components. This is asserting that, given a certain set of inputs, the entire system or a specific component responds within the bounds of the specification.

20: Mulazzani 1985, *Reliability Versus Safety*.

Determinism is a desirable property for designing CPS, in particular, when assessing safety and reliability properties. Determinism, however, is a subtle concept.[21] According to E. A. Lee, "determinism is a property of models, not of physical systems." This property is defined as follows:

> A model is deterministic if given all the *inputs* that are provided to the model, the model defines exactly one possible *behavior*.[21]

21: E. A. Lee 2021, *Determinism*.

In order to assess whether a given model is deterministic or not, we also need to define *input* and *behavior*. For the scope of this thesis, we will define inputs as timestamped messages that may originate from user interactions, from a sensor that perceives the environment or from other systems sending messages over a network. We define behavior as the observable sequence of actions performed by a component in response to its input.

This thesis argues that the safety and reliability requirements of modern complex CPSs can only be achieved if we, at least partially, rely on deterministic models. Since deterministic models define a unique behavior for any set of inputs, this behavior is repeatable and testable. If the component under test behaves deterministically, we can be assured that the tested behavior is identical to the behavior of the component in deployment when posed with the same inputs. Since a deterministic model uniquely defines the correct behavior, we can also detect deviations from this correct behavior and treat them as faults. Commonly, such a fault implies that an assumption made by the physical realization of the behavior was violated.

When using nondeterministic models, however, the number of possible behaviors may grow exponentially as we add components and increase the system complexity. Typically, it is infeasible to reason about all possible behaviors and assess their correctness. However, nondeterministic models are useful in certain cases. For instance, modeling physical processes often requires nondeterministic models. Moreover, for some problems, there is no single correct behavior and a range of behaviors would be acceptable. In such cases, enforcing determinism by reducing the set of acceptable behaviors to a single correct behavior might come at a high cost (e.g., increased latency).

In this thesis, we require that a useful model for CPS design be deterministic. This ensures repeatability of behavior and improves testability, which in turn improves safety and reliability. For some problems, however, it might be useful if the model also allows for deliberately expressing nondeterministic behavior where it is required and understood to do no harm.

1.2.3 Reactivity and Adaptivity

All CPSs are *reactive* systems. Waxman et al. define reactive systems as follows:

> A reactive system is one that is in continual interaction with its environment and executes at a pace determined by that environment.[22]

22: Waxman et al. 1996, *High-Level System Modeling*.

The reactivity results from the interaction of computation with the physical environment. Since many potential events in the environment are not foreseeable, the computation needs to react to external events when they occur.

Reactivity imposes some challenges for modeling the software of CPSs. In particular, a useful MoC must allow the system to react to spontaneous events in the environment. In this thesis, we call a model *reactive* if it includes the notion of sporadic inputs and allows specifying reactions to those inputs.

In addition to reactivity, CPSs are also often required to be *adaptive*. The physical environment is dynamic, and conditions change constantly. A CPS must continuously adapt to these changing conditions. There is a wide range

of possibilities for modeling adaptive behavior. If we can enumerate all possible conditions of the environment that are relevant to a given problem, then we can identify the state in which the environment is in and use conditionals in the program logic. Alternatively, if the underlying MoC supports it, we could dynamically adapt the structure of the program by activating or deactivating certain software components when certain conditions hold. Finally, we could use machine learning and other adaptive strategies to cope with a possibly infinite set of environmental conditions that the system could be faced with. Adaptivity blurs the boundary between *design time* and *runtime*. While adaptivity can be added on the application layer, ideally a MoC also supports dynamic adaptation so that the program structure itself can adapt to changing requirements.

1.2.4 Scalability

The complexity of modern CPSs is increasing rapidly. There are many sources of complexity. Software complexity, for instance, can be measured in lines of code (LOC). Volvo reported that their cars in the year 2020 are estimated to have about 100 million LOC, which is enough to fill a small library.[23] This number is expected to grow by an order of magnitude every 10 years.

23: Antinyan 2020, *Revealing the Complexity of Automotive Software*.

Architectural complexity is also increasing rapidly. This is not just driven by an increasing number of hardware components but also by an increasing heterogeneity. Motivated by the end of Moore's law,[24] hardware designers meet the ever-increasing demands for computational power by specializing and optimizing hardware for specific uses. This trend, however, hinders software productivity as programming heterogeneous hardware requires an increasing amount of specialized code.[25]

24: Moore 1965, *Cramming More Components Onto Integrated Circuits*; Sutter 2005, *The Free Lunch Is Over: a Fundamental Turn Toward Concurrency in Software*.

25: Castrillon and Leupers 2014, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*.

Ideally, a well-chosen MoC supports the software designer in managing this complexity. Applications should be able to scale efficiently from a small to a large code base and from a simple to a large distributed architecture. MoCs help in managing the architectural complexity as they provide an abstraction over the concrete physical architecture (cf. Figure 1.1). In addition, for many MoCs we can construct tools that automatically synthesize an optimized realization on a given target architecture.[26]

26: Castrillon and Leupers 2014, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*; Castrillon, Desnos, et al. 2023, *Dataflow Models of Computation for Programming Heterogeneous Multicores*.

In order to support programmers in managing software complexity, useful MoCs also need to provide mechanisms for structuring programs into components or segments. A major factor in scalability is composability. Ideally, a large application can be constructed from smaller components, which in turn might be constructed from even smaller components. Only if composition is well-defined and preserves the properties of subcomponents can we reuse components and scale up complexity without introducing new problems.

In order for a MoC to be scalable, we also need to be able to derive efficient physical realizations, even if the applications are large. Thus, in terms of scalability, we cannot solely consider the rules imposed by the MoC. We have to also consider whether the rules of the MoC can be implemented efficiently on computers and analyze the performance of these implementations to evaluate scalability.

Scalability has many facets. In this thesis, the term scalability summarizes several characteristics including composability, the ability to efficiently utilize parallel resources, the ability to efficiently utilize distributed resources, as well as the ability to partially automate deployment to various hardware architectures in to manage hardware complexity.

1.2.5 Time

CPSs are hybrid systems. They connect discrete dynamics in the cyber with continuous dynamics in the physical. From a mathematical perspective, interfacing discrete and continuous models is challenging. Even when taking a less formal approach, interfacing software with the physical world requires that some properties of the environment be exposed at the software level.

One particular important property of the environment is time, which is commonly neglected by computing abstractions. While the real-time systems community places a strong emphasis on time, the solutions that we know from that domain do not easily scale up to the computational requirements of modern CPS applications. On the other hand, modern, scalable approaches to computation, as known, for instance, in cloud applications and high-performance computing (HPC), fully abstract over the physical aspects of computation. Time is treated merely as an optimization criterion and not as a property of the program itself.

A useful MoC for CPS design should provide a semantic notion of time that exposes to the program when external events were observed by the system and gives control over when a certain action should be performed. A well-defined notion of time can have the additional benefit of providing a mechanism for coordinating computation in distributed systems.[27]

27: Y. Zhao, J. Liu, and E. A. Lee 2007, *A Programming Model for Time-Synchronized Distributed Real-Time Systems*; Corbett et al. 2013, *Spanner: Google's Globally Distributed Database*.

1.3 This Thesis

This thesis argues that, although there is extensive research on CPS design, none of the existing and popular MoCs sufficiently address all the challenges and requirements that were discussed above. Henzinger and Sifakis observe that there is a semantic gap between models and tools used for safety-critical systems and those taking a best-effort approach.

> The two approaches—critical and best-effort engineering—are largely disjoint. This is reflected by the separation between "hard" and "soft" real time. They correspond to different research communities and different practices. Hard approaches rely on static (design-time) analysis; soft approaches, on dynamic (run-time) adaptation. Consequently, they adopt different models of computation and use different execution platforms, middleware, and networks.[28]

28: Henzinger and Sifakis 2006, *The Embedded Systems Design Challenge*.

Henzinger and Sifakis predicted that this gap would widen, which indeed appears to be the case.

Consider Figure 1.2, which provides an overview of the properties of existing concurrent MoCs.[29] Each MoC is positioned on four axes that denote if the MoC is *timed*, *deterministic*, *scalable*, or *reactive*. On each axis, there is a "yes" and a "no" bin. The diagram is inspired by Karnaugh maps and Veitch charts, which are used for visualizing and minimizing boolean functions.[30] The background color of each square denotes how many properties are in the "yes" bin for this particular field. The darker the color, the more properties are fulfilled by the MoC. The listed MoCs are discussed in depth in Chapter 2, and the position of each MoC is explained in Section 2.10.

29: This form of visualizing a property space was suggested to me by Lars Schütze, who created a similar diagram for his thesis.

30: Veitch 1952, *A Chart Method for Simplifying Truth Functions*; Karnaugh 1953, *The Map Method for Synthesis of Combinational Logic Circuits*.

The property space shown in Figure 1.2 clearly illustrates the semantic gap between existing MoCs. There is no model that is deterministic, reactive, scalable, *and* timed. Therefore, CPS designers need to make compromises and select a model that only fulfills the properties that are most relevant to the specific use case. Many of the modern CPS use cases require reactivity

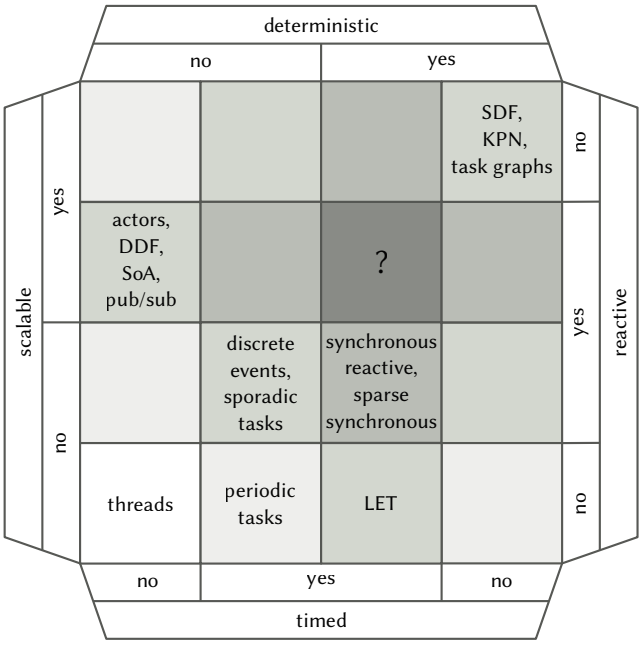


Figure 1.2: Overview of MoCs for cyber-physical systems.

and scalability. However, currently, this can only be achieved by giving up determinism and time predictability.

This thesis aims to close the semantic gap. It establishes a MoC that is deterministic, reactive, scalable, *and* timed. The foundation for this thesis is the reactor model, a novel MoC proposed by Lohstroh, Romeo, et al.[31] The reactor MoC combines aspects of established models into a new paradigm. As is illustrated in this thesis, the reactor model indeed fills the gap identified in Figure 1.2. The core contribution of this thesis is the design, implementation, and evaluation of a technology stack that facilitates the development of efficient and scalable reactor programs and is applicable to a wide range of problems.

31: Lohstroh, Romeo, et al. 2019, *Reactors: A Deterministic Model for Composable Reactive Systems*; Lohstroh 2020, *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*.

1.3.1 Outline

Following the introduction, Chapter 2 surveys a wide range of existing MoCs that are actively used for CPS design, both in academia and industry. This background chapter introduces each MoC in detail and discusses advantages as well as problems and limitations. This discussion also includes an introduction to two popular CPS frameworks: ROS and AUTOSAR. The chapter is concluded by an in-depth discussion of Figure 1.2, which also provides the motivation for the remainder of the thesis.

Chapter 3 introduces the novel reactor model as proposed by Lohstroh, Romeo, et al. This introduction is followed by a discussion of one of the first implementations of the reactor model. In addition, Section 3.3 introduces the DEAR framework, which integrates the reactor model with the automotive standard AUTOSAR. The chapter is concluded by a case study that illustrates how the reactor model as implemented in the Discrete Events for AUTOSAR (DEAR) framework can eliminate nondeterministic behavior in AUTOSAR. This also presents the first demonstration of distributed execution based on the reactor model.

Building on the initial implementation of the reactor model, Chapter 4 introduces the polyglot coordination language Lingua Franca (LF). Lingua Franca implements the reactor model and provides programmers with a language

for expressing reactor programs. It allows for incorporating logic in a target programming language, and the LF compiler automatically transforms LF constructs into the target language. Chapter 4 introduces the syntax of LF, provides various code examples, discusses the execution behavior of LF programs, and illustrates how the execution of LF can be coordinated in distributed systems.

Chapter 5 focuses on the scalability of reactor programs in LF. First, the chapter discusses some limitations of the syntax and the implementation introduced in earlier chapters. Then, it demonstrate how these limitations can be overcome. The main part of Chapter 5 presents a thorough performance evaluation, which compares the performance of LF programs to popular actor frameworks.

Following this evaluation, Chapter 6 discusses some additional limitations of the currently used execution strategy. It presents a potential solution that is called *scheduling enclaves* and allows for partitioning LF programs.

Chapter 7 focuses on design space exploration (DSE), a technique that is commonly used to explore large design spaces and which is commonly applied to optimize program execution on heterogeneous hardware. The chapter introduces a novel DSE framework called Mocasin and discusses potential strategies for optimizing and partitioning LF programs.

Finally, Chapter 8 summarizes related work, and Chapter 9 concludes this thesis. This also includes a discussion of potential future work.

1.3.2 Contributions

Concretely, this thesis makes the following main contributions:

- A detailed survey of existing MoCs that are applied for CPS design (Chapter 2).
- A full implementation of reactors in C++ that includes an efficient runtime (Chapter 3).
- An integration of reactors with the automotive standard AUTOSAR and a case study that illustrates the applicability of the reactor model to distributed industrial use cases (Chapter 3).
- The design and implementation of Lingua Franca (LF), a polyglot coordination language that builds on top of the reactor model (Chapter 4).[32]
- A language extension that allows expressing large-scale reactor programs, optimizations of the C++ reactor runtime that allow for efficient parallel execution, and an extensive performance evaluation (Chapter 5).
- A novel technique called *enclaves* for partitioning LF programs that addresses some limitation of the current reactor implementations (Chapter 6).
- A flexible DSE tool and research framework for mapping dataflow applications to heterogeneous many-cores that can, in principle, also be leveraged for analyzing and mapping LF programs (Chapter 7).

32: Lingua Franca is a highly collaborative project with many contributors. Thus, the contribution is shared between several authors. This thesis introduces both the shared core concepts of LF and the additional contributions of the author.

1.3.3 A Note on Originality

Most of the contributions presented in this thesis result from collaborative work. I believe that truly remarkable research results can only be achieved through collaboration and exchange of ideas among diverse groups of people. This, however, also means that results and ideas often cannot be traced back to an individual author.

The main focus of this thesis lies in contributions that result from my own work. A complete discussion of my work, however, is not possible without presenting the ideas of others and the results of joint work. I took care to indicate when an idea is not my own or when the presented material appeared previously in joint publications. If in doubt, any idea or result presented in this thesis that appeared in prior publications is also due to my coauthors.

Models of Computation for Cyber-physical Systems

2

This chapter surveys existing models of concurrent computation. These are MoCs that explicitly introduce the notion of concurrency and are not limited to strictly sequential execution. This survey focuses in particular on models that are applied in the context of CPS design. Each model is discussed in terms of its potential for addressing the challenges identified in Section 1.2. The discussion also includes programming languages, concepts, and frameworks that are not described by formal modals. Section 2.10 concludes the chapter by summarizing the findings and motivating the need for a new MoC for CPS design.

2.1 Threads

Threads are likely the most common abstraction used to model concurrent programs. A thread refers to the execution of a single, sequential program. As shown in Figure 2.1, multiple such threads of computation form a process and operate concurrently while sharing access to the same memory. This allows one thread to read variables written by another thread within the same process. Commonly, a process starts with one initial thread of computation, which may fork into additional threads. Once a thread has completed its computation, it can be joined by the thread that created it.

Threads are a useful model as they closely capture how processors execute sequential computation and how they may interrupt one thread of computation and continue with another concurrent thread, or how multiple concurrent threads may be executed in parallel on multiprocessors.

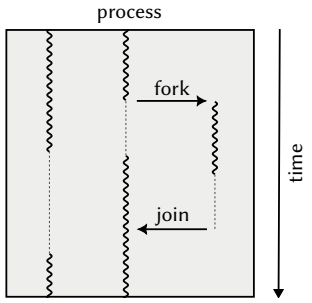


Figure 2.1: A process consisting of multiple threads.

While threads are useful for describing *how* concurrent computation can be performed on potentially parallel processors, they only provide limited control over *when* and *where* the computation is performed or how multiple threads interact. This gives rise to the problem of *data races*. Threads operate on shared memory, but they are treated like individual sequential programs, and by default, there is no orchestration between threads and no predefined order of memory accesses. Depending on when one thread writes a variable, another thread reading this variable might observe different values. For this reason, multithreaded programs are inherently nondeterministic. It is considered the job of the programmer to prune this nondeterminism.[1]

Races for data and other shared resources can be avoided by using various synchronization methods such as mutual-exclusion locks, semaphores or monitors.[2] However, as Edward A. Lee argues, programs that use such techniques are rarely understandable.[1] Moreover, such synchronization methods create yet another problem: the risk of introducing deadlocks. Threads deadlock if each thread holds a resource that another thread needs to progress in its computation. For reasonably sized multithreaded programs, it becomes impossible to decide whether they are free of concurrency bugs.[1] Even rigorous testing does not allow one to conclude that a program is free of concurrency bugs, as the faulty behavior might be unlikely to occur and might not surface in a test execution. So-called *Heisenbugs* are particularly challenging to analyze, debug and reproduce.[3]

1: E. A. Lee 2006, *The Problem With Threads*.

2: Dinning 1989, *A Survey of Synchronization Methods for Parallel Computers*.

3: Musuvathi et al. 2008, *Finding and Reproducing Heisenbugs in Concurrent Programs*.

Research has made significant advances in automatically detecting and even fixing concurrency bugs in multithreaded programs.[4] There are also verification techniques based on Concurrent Separation Logic that may reason about the correctness of multithreaded programs.[5] Moreover, there is a family of techniques called deterministic multi-threading (DTM) that promise a deterministic execution of multithreaded programs even in the presence of concurrency bugs. DTM libraries such as DThreads[6] or Consequence[7] automatically enforce a total order for concurrent store operations. This typically comes at the cost of a significant performance hit, as the technique reduces the exploitable parallelism and introduces additional synchronization points. However, Recent work has made considerable progress in avoiding the bottlenecks of conventional DTM techniques.[8]

Even given the advances in detecting concurrency bugs or ensuring deterministic multithreaded execution, the model of threads remains mostly useful for modeling *how* a concurrent program executes at a low level of abstraction. Threads do not provide a convenient and portable way to reason about *when* and *where* concurrent parts of a program are executed or how they interact. This, however, is paramount for cyber-physical systems.

4: Hong and M. Kim 2015, *A Survey of Race Bug Detection Techniques for Multithreaded Programmes*; Murillo et al. 2014, *Automatic Detection of Concurrency Bugs through Event Ordering Constraints*; Z. Liu et al. 2021, *Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems*.

5: Brookes and O'Hearn 2016, *Concurrent Separation Logic*.

6: T. Liu, Curtsinger, and Berger 2011, *Dthreads: Efficient Deterministic Multithreading*.

7: Merrifield, Devietti, and Eriksson 2015, *High-Performance Determinism with Total Store Order Consistency*.

8: Merrifield, Roghanchi, et al. 2019, *Lazy Determinism for Faster Deterministic Multithreading*.

2.2 Task Models

The concept of tasks is closely related to threads, and the two terms are sometimes used interchangeably. Here, we use thread to denote the execution of a sequential program and task to denote a unit of work. Concretely, we use the definition given by Thoman et al.:[9]

> A *task* is a sequence of instructions within a program that can be processed concurrently with other tasks in the same program. The interleaved execution of tasks may be constrained by control- and data-flow dependencies.

This general definition indicates that tasks denote units of computation that can be composed to form a program. Depending on the precise task model, additional constraints may be defined that control when tasks execute. The task model is widely adopted in various domains, ranging from embedded systems with hard real-time requirements to massively parallel HPC workloads. A scheduler manages the program's execution and decides when and where a task instance is executed. In the following, we will briefly survey a selection of task models and their constraints.[10]

9: Thoman et al. 2018, *A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing*.

10: This overview draws inspiration from Sha, Abdelzaher, et al. 2004, *Real Time Scheduling Theory: a Historical Perspective*; Thoman et al. 2018, *A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing*; Tang, Guan, and Yi 2020, *Real-Time Task Models*.

2.2.1 Periodic and Sporadic Real-time Task Models

Task models have a long tradition in embedded and real-time systems. The most basic model is the periodic real-time task model that was initially proposed by C. L. Liu and Layland.[11] In this model, a program is a set of periodic tasks, where each individual task is characterized by a period and its worst-case execution time (WCET). Each task is executed repeatedly in an infinite sequence, and such an execution of a task is commonly called a *job*. In the periodic task model, the initial task is *released* at an arbitrary initial time, and subsequent tasks are released at fixed intervals, as indicated by the period in relation to the previous job's release time. Each job must be completed before the release of the next, and therefore, the period also defines the *relative deadline* for each job. The *absolute deadline* is given by the release time of the next job.

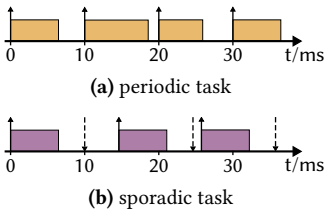


Figure 2.2: A periodic and a sporadic task. A solid arrow represents the release time, and a dashed arrow represents the deadline of a job.

11: C. L. Liu and Layland 1973, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*.

While the periodic task model allows for a trivial feasibility analysis and static derivation of schedules, it is also inflexible as it assumes fixed periods and deadlines. The *sporadic* task model is a generalization of the periodic model that provides more flexibility in the way release patterns and deadlines can be characterized.[12] A sporadic task is characterized by three parameters: the minimum period, the WCET, and the relative deadline.[13] In contrast to the periodic task model, the period is given as a lower bound that characterizes the minimum interval between two job releases of the same task. The relative deadline is given as an additional parameter and is fully independent of the job's period. Figure 2.2 compares possible schedules for the execution of a periodic and a sporadic task.

The literature describes various extensions to the periodic and sporadic task models. This includes *jitter*, which models that jobs are not released precisely and that the actual release time may vary,[14] as well as *bursts*, which model that a series of jobs may arrive at an interval less than the period due to jitter.[15] Other task models use a stochastic approach to model varying arrival rates.[16] In the offset-based task model, each task consists of several subtasks that are characterized by an offset.[17] Each subtask is released with the given offset relative to the release time of the outer task.

12: S. K. Baruah, Mok, and Rosier 1990, *Preemptively Scheduling Hard-real-time Sporadic Tasks on one Processor*; Sprunt, Sha, and Lehoczky 1989, *Aperiodic Task Scheduling for Hard-Real-Time Systems*.

13: Mok 1983, *Fundamental Design Problems of Distributed Systems for the Hard-real-time Environment*.

14: Audsley et al. 1993, *Applying New Scheduling Theory To Static Priority Pre-Emptive Scheduling*.

15: Tindell, Burns, and A. J. Wellings 1994, *An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks*.

16: Atlas and Bestavros 1998, *Statistical Rate Monotonic Scheduling*.

17: Palencia and Gonzalez Harbour 1998, *Schedulability Analysis for Tasks with Static and Dynamic Offsets*.

2.2.2 Graph-based Task Models

The periodic and sporadic task models and their various extensions are rather inflexible in modeling varying workloads. The execution time of a task might depend on the state of the system or on the input data itself. A common example of a data-dependent task is MPEG decoding.[18] MPEG distinguishes various frame types, some of which are more complex to decode than others. In the periodic task model, the WCET always needs to account for the worst case, which leads to a pessimistic schedulability analysis.

Graph-based task models solve this problem by allowing different jobs of the same task to have different periods and relative deadlines. A graph structure models which job of a task has which parameters. For instance, the generalized *multiframe* model characterizes a task as a cyclic graph.[19] Each node represents a job configuration and is annotated with the WCET and the relative deadline of the job. The edges in the graph are annotated with the period (or inter-release time). The execution first releases an initial job and then cycles through the graph, changing the configuration for each subsequent job. Figure 2.3 shows an example multiframe task. The multiframe model has been extended to also support arbitrary non-cyclic graphs.[20] If a node has multiple outgoing edges, then any of the outgoing edges can be chosen when releasing the next job.

There is also a family of *branching* task models that have less focus on data dependence and focus more on control flow patterns, where branches may influence workload release patterns.[21] In the most general form, the possible release patterns of a task are described as an arbitrary directed graph[22] or as a task automaton.[23] Graph-based task models allow an application to adapt to its input or to the system's state. However, this adaptivity is limited to statically known configurations and transitions.

18: Mok and Chen 1997, *A Multiframe Model for Real-Time Tasks*.

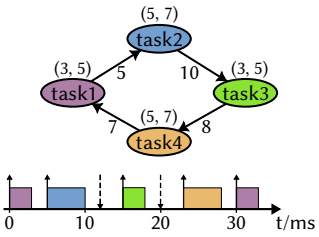


Figure 2.3: A multiframe task and a corresponding release pattern.

19: S. K. Baruah, Chen, et al. 1999, *Generalized Multiframe Tasks*.

20: Tchidjo Moyo et al. 2010, *On Schedulability Analysis of Non-cyclic Generalized Multiframe Tasks*.

21: S. K. Baruah 1998, *Feasibility Analysis of Recurring Branching Tasks*; Anand et al. 2008, *Compositional Feasibility Analysis of Conditional Real-Time Task Models*; S. K. Baruah 2010, *The Non-cyclic Recurring Real-Time Task Model*.

22: Stigge et al. 2011, *The Digraph Real-Time Task Model*.

23: Fersman, Pettersson, and Yi 2002, *Timed Automata with Asynchronous Processes: Schedulability and Decidability*.

2.2.3 Parallel Task Models

In order to utilize parallel hardware, there are also a range of task models that allow for expressing parallelism within a task. In the gang task model, for instance, each task may consist of multiple parallel threads that execute

simultaneously.[24] The fork/join task model characterizes a task as a sequence of segments, where odd segments execute sequentially and even segments execute in parallel using multiple threads.[25] The parallel synchronous task model generalizes this pattern and allows for an arbitrary number of threads in each segment.[26] An example parallel synchronous task is shown in Figure 2.4. Such patterns are typical for tasks generated from parallel for loops, as used in languages like OpenMP[27] and Cilk Plus.[28] This model was further generalized into the directed acyclic graph (DAG) task model, where a DAG describes the precedence relations of threads within a task.[29]

24: Feitelson and Rudolph 1992, *Gang Scheduling Performance Benefits for Fine-Grain Synchronization*; Dong and C. Liu 2017, *Analysis Techniques for Supporting Hard Real-Time Sporadic Gang Task Systems*.

25: Lakshmanan, Kato, and Rajkumar 2010, *Scheduling Parallel Real-Time Tasks on Multi-core Processors*.

26: Saifullah et al. 2011, *Multi-core Real-Time Scheduling for Generalized Parallel Task Models*.

27: Ayguade et al. 2009, *The Design of OpenMP Tasks*.

28: Robison 2013, *Composable Parallel Patterns With Intel Cilk Plus*.

29: S. Baruah et al. 2012, *A Generalized Parallel Task Model for Recurrent Real-time Processes*; Saifullah et al. 2011, *Multi-core Real-Time Scheduling for Generalized Parallel Task Models*.

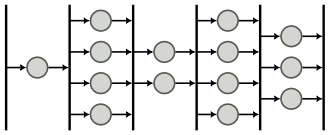


Figure 2.4: A parallel synchronous task.

2.2.4 Limitations

The main motivation for developing various real-time task models is to reason about what is computable within a given time. Concretely, real-time system researchers and their modeling efforts are driven by the question: Is there a *feasible* schedule for a given set of tasks? A feasible schedule denotes a schedule that allocates tasks to a set of processors such that all deadlines are met. Research in the field of real-time scheduling has a long history and has not only produced a wide range of models but also various scheduling algorithms.[30]

30: Sha, Abdelzaher, et al. 2004, *Real Time Scheduling Theory: a Historical Perspective*.

While periodic, sporadic, and related task models are useful for schedule feasibility analysis, they are less useful as a programming abstraction that helps humans and compilers reason about the interaction of concurrent tasks. The discussed task models fall short on capturing data dependencies between tasks and do not provide mechanisms for coordinating access to shared resources. Similar to threads, the programmer of a task needs to ensure that reads and writes of data, as well as other accesses to shared resources, are protected by synchronization primitives like locks. This is problematic not only because synchronization is difficult to get right, but also because it introduces the problem of priority inversion.[31] In a system with limited shared resources, a low-priority job that holds a resource might block a higher-priority job that acquires the same resource.

31: Sha, Rajkumar, and Lehoczky 1990, *Priority Inheritance Protocols: an Approach To Real-Time Synchronization*.

Since common real-time task models do not capture data dependencies between tasks, the program's behavior may depend on the concrete schedule. If the release time or execution time of tasks varies, this may influence the result of the computation. Consider the example given in Figure 2.5. There are three periodic tasks: A, B, and C. The deadline of each task is equal to its period. The execution time varies between jobs,[32] and also the time at which Task B is scheduled for execution varies. Each of the tasks reads its input at the beginning of its execution and writes the results at the end. This is denoted by the black lines at the beginning and end of a job in Figure 2.5.

32: The execution time could vary because the amount of computation performed by the task is data-dependent or because of timing variations in the underlying hardware, e.g., due to caches.

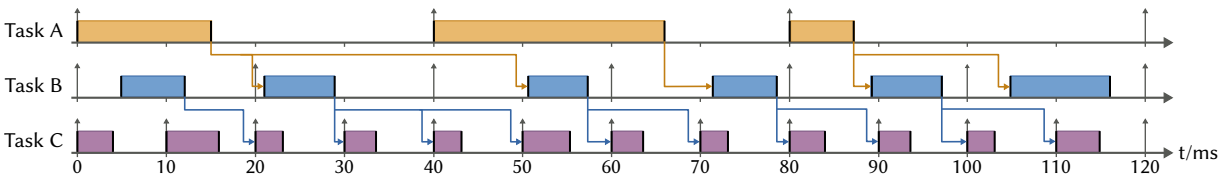


Figure 2.5: An example schedule for 3 tasks that illustrates the influence of jitter in execution and release times on the data dependencies between tasks. This figure is loosely based on Figure 3 in Gemlau et al. 2021, *System-Level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-Time Automotive Software*.

Task B reads the output of Task A, and Task C reads the output of Task B. The concrete data path, however, varies between hyperperiods.[33] Which job reads the output of which job is not determined by the program but solely by the concrete schedule and the interleaving of tasks. Thus, basic task models

33: Intervalls of 40 ms in this example.

expose the same nondeterminism that is inherent to threads. Therefore, they have limited usability for modeling and reasoning about complex systems without additional constraints.

In order to capture the data dependencies between tasks, they can also be arranged in a DAG. Task graphs are particularly popular in HPC applications.[34] However, task graphs typically do not have any real-time constraints. Scheduling algorithms in HPC are primarily concerned with maximizing throughput and minimizing makespan, but they do not provide control over when or where a task gets executed. Therefore, task graphs are not particularly useful for CPS design. Moreover, the DAG structure does not permit any feedback loops, which are a fundamental requirement for many control applications.

34: Kwok and Ahmad 1999, *Static Scheduling Algorithms for Allocating Directed Task Graphs To Multiprocessors*; Thoman et al. 2018, *A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing*.

2.3 The Actor Model

The Hewitt actor model is a model of concurrent computation that was originally proposed by Carl Hewitt[35] and later refined by Gul Agha.[36] In this model, actors are the universal building blocks for concurrent computation. Each actor has private state, a mailbox for receiving messages, and a behavior. The behavior is invoked in response to a message received in an actor's mailbox. An actor's behavior may:

- perform computation,
- modify the local actor state,
- send new messages,
- create new actors,
- or adapt the behavior that is invoked for the next message.

35: Hewitt, Bishop, and Steiger 1973, *A Universal Modular ACTOR Formalism for Artificial Intelligence*; Hewitt 1977, *Viewing Control Structures as Patterns of Passing Messages*.

36: Agha et al. 1997, *A Foundation for Actor Computation.*

In contrast to threads and most task models, the actor model introduces a notion of local, encapsulated state that is similar to the private state of objects in object-oriented programming.[37] Moreover, the behaviors of each actor are mutually exclusive. Thus, only one behavior has access to the actor's state at a time. The encapsulation of state in combination with the mutual exclusion of behaviors effectively prevents low-level data races that may occur when using threads or tasks.

37: Stroustrup 1988, *What Is Object-Oriented Programming?*

Unlike threads and tasks, actors are reactive. While threads are continuously executing and tasks are executed according to a (more or less) fixed schedule, the behaviors of actors are triggered by the messages that they receive. Actors react to the actual workload, and the behaviors are processed as needed. There is no additional knowledge needed to execute a behavior. Since the behaviors of different actors are fully independent, they can be executed in parallel. And since the there is no shared memory in the actor model, actors can also be easily distributed in a networked system.

The actor model allows programmers to express concurrent computation in seemingly simple terms while avoiding the most common pitfalls of threads and tasks. In addition, the actor model promises to allow efficient implementations of actor programs that can exploit parallel hardware and transparently scale to distributed systems. Due to these characteristics, the actor model has become widely accepted and deployed in academia, as well as in production.

2.3.1 Actor Languages and Frameworks

There is a wide range of programming languages and frameworks implementing the actor model (or variants thereof). Early implementations include Act,[38] Cantor,[39] and Rosette.[40] Erlang is one of the first commercially successful actor-based languages and has been widely used for telecommunication applications in the 90s.[41] More modern adaptations of the actor model include the languages Elixir,[42] P,[43] and Pony[44] as well as the actor frameworks Actix,[45] Akka,[46] C++ Actor Framework (CAF),[47] and Ray.[48] Akka and Ray in particular are very successful and widely used in industry.

Actor frameworks like Akka, CAF and Ray make the actor model available in mainstream programming languages like Java, C++ and Python. While this makes it easier for users to adopt the actor model, it also bears the risk that users mix concurrency models or violate actor semantics.[49] In particular, the aforementioned languages cannot prevent access to shared memory. The Actix actor framework leverages the strong type system of Rust to have more fine-grained control over state access. Pony is a dedicated actor language and leverages a strong type system that builds on similar ideas as Rust's type system, to prevent shared state between actors. Some actor languages, like Rebeca, also integrate formal verification methods to ensure the correctness of actor programs.[50]

38: Lieberman 1987, *Concurrent Object-Oriented Programming in Act 1*.

39: Athas and Boden 1988, *Cantor: an Actor Programming System for Scientific Computing*.

40: Tomlinson et al. 1988, *Rosette: an Object-Oriented Concurrent Systems Architecture*.

41: Virding et al. 1996, *Concurrent Programming in Erlang (2nd Ed.)* Armstrong 2007, *A History of Erlang*.

42: D. Thomas 2018, *Programming Elixir 1.6: Functional|> Concurrent|> Pragmatic|> Fun*.

43: Desai et al. 2013, *P: Safe Asynchronous Event-Driven Programming*.

44: Clebsch et al. 2017, *Orca: GC and Type System Co-Design for Actor Languages*.

45: The Actix Team 2023, *Actix: Actor Framework for Rust*.

46: Roestenburg, Williams, and Bakker 2016, *Akka in Action*.

47: Charousset, Hiesgen, and T. C. Schmidt 2016, *Revisiting Actor Programming in C++*.

48: Moritz et al. 2018, *Ray: A Distributed Framework for Emerging AI Applications*.

49: Tasharofi, Dinges, and Johnson 2013, *Why Do Scala Developers Mix the Actor Model with other Concurrency Models?*

50: Sirjani, Movaghar, et al. 2004, *Modeling and Verification of Reactive Systems Using Rebeca*; Sirjani and Jaghoori 2011, *Ten Years of Analyzing Actors: Rebeca Experience*.

51: Torres Lopez et al. 2018, *Programming with Actors: State-of-the-Art and Research Perspectives*.

2.3.2 Limitations

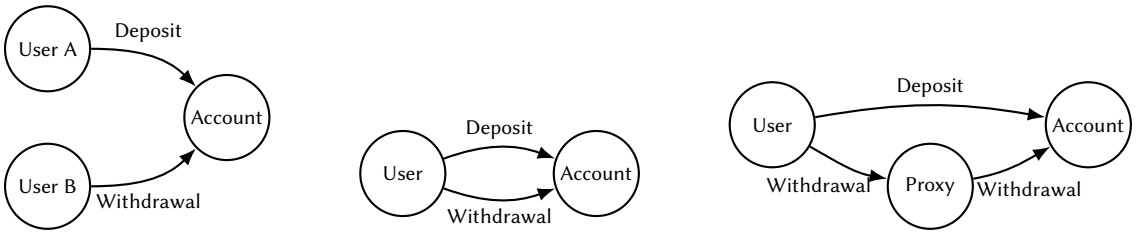
This subsection uses examples and arguments previously published in Menard, Lohstroh, et al. 2023, *High-Performance Deterministic Concurrency Using Lingua Franca*.

The actor model is widely accepted and deployed in production for its promise to allow programmers to easily express concurrency, provide high execution performance, and scale well to large datasets and complex applications while preventing *low-level* data races. However, actors still expose nondeterminism in the form of *high-level* data races,[51] a problem that becomes considerably more challenging to manage as the complexity of a program grows.

Bank Account Example

Consider the simple example in Figure 2.6a. The `Account`actor manages the balance of a bank account that two users interact with. `User A`sends a deposit message, increasing the account's balance, and `User B`sends a withdrawal message, decreasing the account's balance. If we assume that the balance is initialized to 0 and the account only grants a withdrawal if the resulting balance is not negative, then there are two possible behaviors. If A's message is processed first, then the withdrawal is granted to B. If B's message is processed first, then the withdrawal is denied. The actor model assigns no meaning to the ordering of messages. Therefore, there is no well-defined correct behavior for this example.

The reader may object that for an application like that of Figure 2.6a, the order of transactions is intrinsically nondeterministic, and any additional nondeterminism introduced by the software framework is inconsequential. However, if we focus on testability, we see that even identical inputs can yield different results, making testing more difficult. If we focus on consistency, the problem that different observers of the same events may see different behaviors becomes problematic. In databases, it is common to assign time stamps to external inputs and then treat those timestamps as a semantic



(a) Deposit and Withdrawal sent by different users.

(b) Deposit and Withdrawal sent by same user.

(c) Withdrawal sent via a proxy.

Figure 2.6: Example actor programs that may expose nondeterministic behavior.

property of the inputs and define the behavior of the database relative to those time stamps. This perspective is in line with our definition of determinism in Section 1.2.2. If we define *inputs* in Figure 2.6a to be time-stamped user queries and *behavior* to be the sequence of actions taken by the Account, then it is reasonable to demand determinism.

Consider Figure 2.6b, which has only one user. Even if this one user first sends a deposit and then a withdrawal message, the actor model does not guarantee that the receiving actor sees and processes the incoming messages in this order. While some actor frameworks, e.g., Akka and Erlang, guarantee in-order message delivery, others, e.g., AmbientTalk,[52] expressly do not. Yet, even if the framework guarantees point-to-point in-order message delivery, this property is not transitive. If we add a proxy, as shown in Figure 2.6c, then we cannot make any assumptions about the order in which `Account` receives messages. This example further illustrates that composing actors can have unexpected side effects.

52: Van Cutsem et al. 2014, *AmbientTalk: Programming Responsive Mobile Peer-To-Peer Applications With Actors*.

Aircraft Door Example

The consequences of nondeterministic behavior can be fatal for CPSs. Consider, for example, an aircraft door. Passenger aircraft are equipped with emergency escape slides that automatically deploy when the corresponding door of the aircraft is opened. The deployment mechanism needs to be explicitly disarmed in order to safely open the doors in a regular parking position with a passenger ramp.

Let us assume that the door is controlled by an embedded control component that resides inside the door and communicates with other components via a network. The door controller receives two signals that can be sent from the cockpit: `disarm`for disarming the emergency escape slides and `open`for opening the door. We further assume that the `disarm`signal is intercepted by another sensor component that performs a safety check. Only if a passenger ramp is indeed placed in front of the door will the sensor pass the `disarm` signal on to the door. Figure 2.7 shows a possible actor realization of this example. This program exposes the same nondeterministic behavior as the program in Figure 2.6c. Consequently, we cannot make any assumptions about the order in which the messages arrive at the door. If we model the application using actors without introducing additional protocols or coordination mechanisms, then it would be up to chance whether the emergency slides deploy in the parking position.

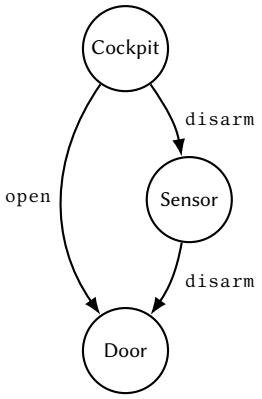


Figure 2.7: An actor implementation of the aircraft door example.

Discussion

Implementing solutions to practical concurrency problems with actors can be challenging. Even seemingly simple concurrency problems like the ones discussed above require high programming discipline, and solutions are typically difficult to maintain and tend to lack modularity. In addition, the inherent nondeterminism of actor frameworks makes it challenging to verify such solutions. Erroneous behavior might only occur in a fraction of executions, and thus integration tests cannot reliably detect such *Heisenbugs*.[53]

In a recent study, Bagherzadeh et al. analyzed bugs in Akka programs that were discussed on StackOverflow or GitHub and determined that 14.6% of the bugs were caused by races. This makes high-level races the second most common cause of bugs in Akka programs, after errors in the program logic.[54] In a similar study of 12 actor-based production systems, Hedden and X. Zhao determined that 3.2% of the reported bugs were caused by bad message ordering, 4.8% were caused by incorrect coordination mechanisms, 4.8% were caused by erroneous coordination at shutdown, and 2.4% of bugs were caused by erroneous coordination at startup.[55] Note that these numbers only cover *known* bugs in their studied projects, and, as noted by the authors, the majority of the reported message ordering bugs belonged to the Gatling project because it already incorporated a debugging tool called Bita[56] that is designed to detect such bugs. It is reasonable to suspect that there are more undetected bugs in projects that do not use specialized debugging tools.

The actor community has addressed the inherent nondeterminism of actors and the resulting bugs by introducing better tools for analyzing and debugging actor programs. This includes TransPDOR,[57] Bita,[56] Actoverse,[58] iDeA,[59] CauDEr,[60] and Multiverse debugging.[61] The Voyager tool shown in Figure 2.8, which is part of the Multiverse debugging approach, can visualize the multitude of possible actor program behaviors. Users can interactively expand the graph and execute various queries.

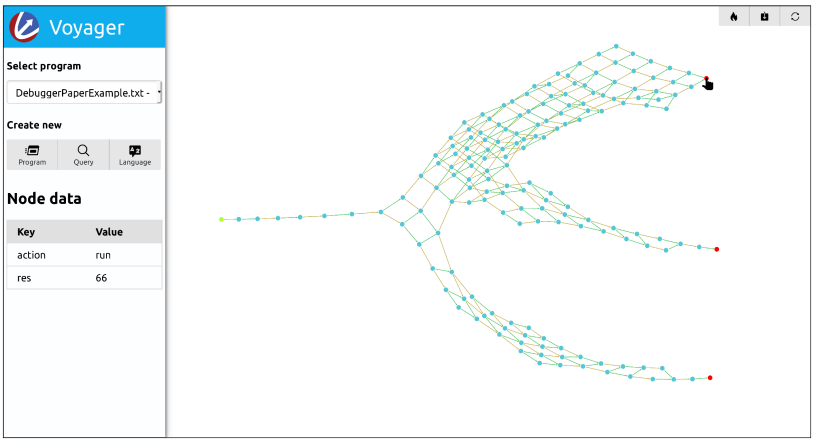


Figure 2.8: The Voyager tool for debugging the multitude of possible behaviors of actor programs. This image is reproduced from Torres Lopez et al. 2019, *Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper)* licensed under CC-BY 3.0.

While the tools mentioned above are valuable solutions, this thesis argues that a concurrent programming model should not rely on the users to analyze programs and prune unintended nondeterminism, in particular in the context of CPSs, where the consequences of nondeterministic behavior could be fatal. A safe concurrent programming model should provide deterministic semantics by default and allow the programmer to introduce nondeterminism only where it is desired and understood to do no harm. In such cases, the aforementioned tools for analyzing nondeterministic behavior can still be utilized to analyze and debug the implementation.

53: Musuvathi et al. 2008, *Finding and Reproducing Heisenbugs in Concurrent Programs*.

54: Bagherzadeh et al. 2020, *Actor Concurrency Bugs: a Comprehensive Study on Symptoms, Root Causes, Api Usages, and Differences*.

55: Hedden and X. Zhao 2018, *A Comprehensive Study on Bugs in Actor Systems*.

56: Tasharofi, Pradel, et al. 2013, *Bita: Coverage-guided, Automatic Testing of Actor Programs*.

57: Tasharofi, Karmani, et al. 2012, *TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs*.

58: Shibanai and Watanabe 2017, *Actoverse: A Reversible Debugger for Actors*.

59: Mathur, Ozkan, and Majumdar 2018, *IDeA: An Immersive Debugger for Actors*.

60: Lanese et al. 2018, *CauDEr: A Causal-Consistent Reversible Debugger for Erlang*.

61: Torres Lopez et al. 2019, *Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper)*.

2.3.3 Related Models and Paradigms

There are a multitude of related models, paradigms, languages, and frameworks that build on similar ideas as the Hewitt actor model. The service-oriented architecture (SoA) paradigm, for instance, views a system as a composition of multiple services, where each service has a well-defined interface and exchanges messages with other services or users using these service interfaces.[62] The services in an SoA are typically monolithic applications. The microservice paradigm breaks those applications down into smaller units, such that each microservice has only a single responsibility.[63] Microservices can be easily understood and independently deployed, scaled, and tested. However, microservices only become useful once they are composed into services and applications, which may easily comprise hundreds of microservices. While each microservice can be easily understood individually, the interaction of hundreds of microservices can be immensely complex and due to the problems discussed above, this interaction is difficult to understand, test, and debug.

62: Perrey and Lycett 2003, *Service-oriented Architecture*; Papazoglou and Heuvel 2007, *Service Oriented Architectures: Approaches, Technologies and Research Issues*; K. B. Laskey and K. Laskey 2009, *Service Oriented Architecture*.

63: Thönes 2015, *Microservices*; Dragoni et al. 2017, *Microservices: Yesterday, Today, and Tomorrow*.

Communication between multiple services is often implemented via remote procedure calls (RPCs).[64] The client code calls a seemingly conventional function, but instead of returning a result immediately, the function relays the arguments to a service and waits for a response, which is then returned. To decouple the execution of client and server, commonly, the RPC does not block until the server responds but instead returns a *future* immediately. A future is a placeholder for the promise that a result will be delivered eventually.[65] The client code calls a get procedure on the future to read the result only when it is needed. If the result is not delivered yet, the get procedure blocks until the corresponding response is received. This process can be seen as a simple request-response protocol on top of actors.

64: Nelson 1981, *Remote Procedure Call*; Birrell and Nelson 1984, *Implementing Remote Procedure Calls*.

65: Baker and Hewitt 1977, *The Incremental Garbage Collection of Processes*.

While hiding the precise mechanics of asynchronous communication may look like a solution for simplifying the interaction of distributed components, programs that use RPCs are still subject to the same concurrency bugs as found in actors. If a certain order of requests or responses is required, this can only be achieved by carefully considering when to call a remote procedure and when to retrieve the result from a future. This semi-transparent approach might give a wrong sense of safety, as is also argued by Tanenbaum and Renesse.[66]

66: Tanenbaum and Renesse 1988, *A Critique of the Remote Procedure Call Paradigm*.

Publish/subscribe is another paradigm that is closely related to actors. The paradigm adds a communication layer that decouples sender and receiver. Instead of sending messages directly to the mailbox of an actor, a publisher sends a message to a topic, which is a named communication channel. Other components may subscribe to the topic in order to receive messages that are published on it.[67] Popular implementations of publish/subscribe include the Message Queuing Telemetry Transport (MQTT)[68] and Data Distribution Service (DDS)[69] standards.

67: Eugster et al. 2003, *The Many Faces of Publish/subscribe*.

68: OASIS 2019, *MQTT Version 5.0*.

69: OMG 2015, *Data Distribution Service*.

There are many more concepts that utilize asynchronous message passing and are closely related to Hewitt actors. There are too many to list and cite them all. However, without additional safeguards, they all share the problems of Hewitt actors and expose nondeterminism, which can be subtle if the paradigm tries to hide the fact that asynchronous communication is involved, like it is the case with RPCs.

2.4 Dataflow and Process Networks

The previous section introduced the actor model as defined by Hewitt. However, for the remainder of this thesis, we will use a broader definition of *actors* given by Lohstroh and E. A. Lee:

> *Actors* are concurrent objects that communicate by sending each other messages.[70]

70: Lohstroh and E. A. Lee 2019a, *Deterministic Actors*.

In the Hewitt actor model, actors use mailboxes and exchange messages asynchronously. As discussed in the previous section, the asynchronous message-passing paradigm makes Hewitt actors inherently nondeterministic. However, there is a wide range of deterministic concurrent MoCs that fall under the broader definition of actors given above. In this section, we discuss two families of actor-based MoCs that are called *process networks* and *dataflow*. Figure 2.9 shows the relationships between the various models that are discussed in the following.

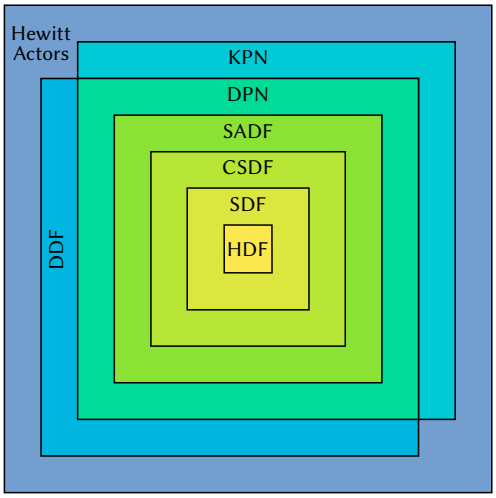


Figure 2.9: Venn diagram highlighting the relationships between various actor-based MoCs. This Venn diagram is based on the one given in Goens 2021, *Improving Model-Based Software Synthesis: A Focus on Mathematical Structures*, p. 104.

2.4.1 Kahn Process Networks

One of the first concurrent MoCs was described by Gilles Kahn and is now known as Kahn process networks (KPNs).[71] Based on Scott's formalism for modeling sequential computations as continuous functions, Kahn created a denotational semantics describing how multiple sequential computations could occur in parallel.[72] In this model, the continuous functions operate on infinite data streams, allowing for data exchange between the functions. KPNs can also be thought of as a set of Turing machines that are connected by infinite one-way tapes.[73]

71: Kahn 1974, *The Semantics of a Simple Language for Parallel Programming*.

72: E. A. Lee and Matsikoudis 2009, *The Semantics of Dataflow With Firing*.

73: Parks 1995, *Bounded Scheduling of Process Networks*.

KPNs are commonly described as directed graphs, such as the one given in Figure 2.10. Each node in the graph represents a process (or actor). Processes execute sequentially and independently of each other. Each process has local state and there is no shared memory. In order to exchange data with other processes, each process may read from or write to a set of predefined data streams called *channels*. In Figure 2.10, the channels are indicated by the directed edges.

Kahn and MacQueen give an implementation of the KPN model that uses blocking read operations. Since this blocking semantics is not a requirement of Kahn's original model, the Kahn-MacQueen implementation is more restrictive.[74] However, the Kahn-MacQueen model is the most common

74: Khasanov, Goens, and Castrillon 2018, *Implicit Data-Parallelism in Kahn Process Networks: Bridging the MacQueen Gap*.

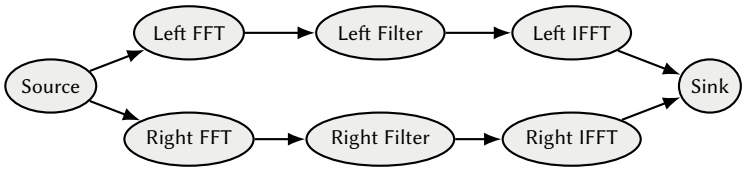


Figure 2.10: A KPN application implementing an audio filter with two channels.

implementation of KPN, and often the blocking reads are implied when the term KPN is used. For the remainder of this thesis, we will use KPN to refer to the Kahn-MacQueen implementation.

As KPN processes can be represented as Turing machines, termination is not decidable. Due to the blocking read operations, KPN processes may deadlock. If, for instance, a process reads from a channel that no other process can write to, then the process deadlocks. For general KPN programs, deadlock freedom is not decidable.[73]

While the Kahn-MacQueen model only requires blocking reads and assumes unbounded buffers, real implementations need to handle bounded memory. Therefore, KPN channels are commonly implemented as bounded first-in, first-out (FIFO) buffers with blocking writes. Consequently, bounding a channel introduces the risk of additional deadlocks. Finding suitable buffer bounds for general KPNs that do not introduce deadlocks is an undecidable problem. However, Parks describes a runtime algorithm that detects deadlocks during execution and resizes buffers where needed.[73] This mechanism is commonly used in KPN implementations and can be further optimized.[75]

75: Geilen and Basten 2003, *Requirements on the Execution of Kahn Process Networks*.

Example Application

The example KPN application in Figure 2.10 implements an audio filter. The source process reads a stereo audio stream with two channels from a file or from an I/O device. It splits the incoming audio data into two channels and forwards chunks of data to the two fast Fourier transformation (FFT) processes. The data chunks are commonly called *tokens*. Tokens are atomic units of data that can be transferred on a channel. Typically, tokens are typed and have a fixed size.

The FFT processes convert the audio stream into the frequency domain. Next, the filter processes apply a filter operation (e.g., a low-pass filter removing low frequencies). The inverse FFT processes convert the signal back to the time domain. Finally, the sink process collects the results, combines the two channels, and writes them to disk.

Listing 2.1 gives a pseudocode implementation of the filter KPN process from Figure 2.10. The process operates in an infinite loop. It simply reads a token from the input channel, applies the filter function, and then writes the result to the output channel. The procedures readToken and writeToken need to be provided by a supporting library that implements the KPN channels. The FFT and inverse FFT processes can be implemented similarly.

```
while true do
    ▷ Read from the input channel. ◁
    data ⟵readToken(in)
    ▷ Apply the filter to the data. ◁
    filteredData ⟵filter(data)
    ▷ Write result to the output channel. ◁
    writeToken(out, filteredData)
```

Listing 2.1: Pseudocode of a KPN process implementing the filter from Figure 2.10.

2.4.2 Dataflow

In the same year in which Kahn published his denotational semantics of process networks, Jack Dennis published a paper that approaches concurrent computation differently. He describes an operational semantics of *dataflow* that is based on the concept of atomic *firings*.[76] Similar to KPN, dataflow programs are defined as directed graphs where the notes are actors and the edges denote communication channels. However, in Dennis' concept of

76: Dennis 1974, *First Version of a Data Flow Procedure Language*; Dennis 1986, *Data Flow Computation*.

dataflow, the actors are not defined as continues processes. Instead, each actor is defined by a set of firing rules and functions that are applied if a rule matches.

Dennis Dataflow

In the generalized formalism given by E. A. Lee and Matsikoudis, $\perp$ denotes an absent token, and $*$ denotes an arbitrary token on a channel.[77] Firing rules are given as tuples, with an entry for each input channel of an actor. The filter process in Figure 2.10 only has a single input and thus would be defined as a dataflow actor using the firing rule $(*)$. When this firing rule is satisfied, i.e., there is at least one token in the input channel, then the actor fires. It consumes the input token, executes the filter function, and produces a new token on the output channel.

77: E. A. Lee and Matsikoudis 2009, *The Semantics of Dataflow With Firing*.

The sink process in Figure 2.10 has two inputs and can be defined as a dataflow actor in multiple ways. Using the firing rule $(*, *)$, the sink actor would fire when a token is present on both inputs. Alternatively, we can use two rules $(*, \perp)$ and $(\perp, *)$. In this case, the actor fires when there is at least one token on one of the input channels. Both rules can be satisfied simultaneously if there is at least one token available in both input channels. However, there is no defined order between simultaneous firings. Hence, this notion of dataflow is nondeterministic. This dynamic variant of dataflow is commonly called Dennis dataflow or dynamic dataflow (DDF).

E. A. Lee and Parks show that an additional condition is sufficient to make the model deterministic. Moreover, they show that this deterministic variant of DDF can be embedded into a KPN model.[78] Therefore, it is commonly referred to as a dataflow process network (DPN).

78: E. A. Lee and Parks 1995, *Dataflow Process Networks*.

In the Venn diagram (Figure 2.9), DPN is denoted as the intersection of KPN and DDF. While E. A. Lee and Parks have proven that DPN is included in KPN, we do not know if all KPN models can be expressed in DPN.[79]

79: Goens 2021, *Improving Model-Based Software Synthesis: A Focus on Mathematical Structures*.

Synchronous Dataflow

A more restrictive form of dataflow called synchronous dataflow (SDF) was introduced by E. A. Lee and Messerschmitt.[80] While this dataflow variant is less expressive than KPN and DDF, it is statically analyzable. Both a schedule and channel bounds can be derived statically.[81] Similar to DDF, SDF actors are defined in terms of firing rules. However, the SDF firing rules precisely define input and output rates, which specify how many tokens the actor consumes or produces on each channel per firing.

80: E. A. Lee and Messerschmitt 1987, *Synchronous Data Flow*.

81: Parks 1995, *Bounded Scheduling of Process Networks*.

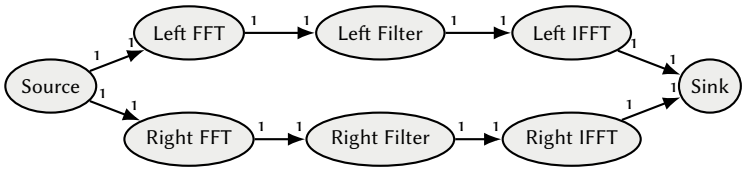


Figure 2.11: An SDF application implementing an audio filter with two channels.

The KPN example in Figure 2.10 can also be expressed as an SDF application. Figure 2.11 shows the audio filter example implemented as an SDF graph with all rates set to 1. This particular variant of SDF is called homogeneous SDF (HSDF).

For some applications, the fixed rates in SDF are too restrictive. They do not allow for any adaptation of the communication patterns at runtime. Many more variants of SDF have been proposed to overcome its limitations.

Particularly noteworthy are cyclo-static dataflow (CSDF) and scenario-aware dataflow (SADF). In CSDF, each actor defines a repeating pattern (cycle) of firing rates.[82] SADF is even more general and allows to dynamically disable or enable certain paths in the dataflow graph.[83] Both models preserve the static analyzability of SDF.

82: Bilsen et al. 1996, *Cyclo-Static Dataflow*.

83: Theelen et al. 2006, *A Scenario-aware Data Flow Model for Combined Long-run Average and Worst-case Performance Analysis*.

2.4.3 Implementations

KPN, SDF and related models have been implemented in a wide range of programming languages and frameworks. PREESM, for instance, is a framework for modeling parameterized and interfaced SDF (πSDF) applications.[84] πSDF is an SDF variant that allows dynamic reconfiguration of the firing rates of each actor.[85]

C for process networks (CPN) is a domain-specific language (DSL) based on C macros that allow to annotate C functions as KPN processes and to instantiate and connect such processes.[86] It also provides specialized directives for declaring SDF actors.

The Ptolemy II framework is a modeling platform that supports a broad selection of MoCs including KPN, SDF and Hewitt actors.[87] Ptolemy II also allows mixing different MoCs through hierarchical abstractions. The CAL actor language (CAL) is a dedicated dataflow language that was developed as part of the Ptolemy II project.[88]

Many more so-called *stream processing* languages that build on the fundamental ideas of dataflow and process networks have been proposed in the literature and are actively used in production.[89] Examples include StreamIt for efficient parallel data processing;[90] OpenStream, which extends OpenMP with dataflow semantics;[91] and the SPL language for big data processing that is part of IBM's Streams product.[92]

Process networks and dataflow provide a powerful abstraction as they fully decouple computation and communication. Compared to Hewitt actors, however, they are more restricted, which makes them both more predictable and analyzable. Yet, they are expressive enough to capture many problems, in particular streaming applications. There is a wide range of toolflows that specialize in KPN or SDF models and their variants and that use DSE techniques to automatically derive efficient and highly parallel implementations on heterogeneous many-cores or in distributed systems.[93] There are also dedicated programming languages designed such that a dataflow graph can be abstracted from an imperative program.[94] Chapter 7 discusses such toolflows in more depth and introduces the Mocasin toolflow, which is a contribution of this thesis.

84: Pelcat et al. 2014, *PREESM: A Dataflow-based Rapid Prototyping Framework for Simplifying Multicore DSP Programming*.

85: Desnos et al. 2013, *PiMM: Parameterized and Interfaced Dataflow Meta-Model for MPSoCs Runtime Reconfiguration*.

86: Sheng et al. 2014, *A Compiler Infrastructure for Embedded Heterogeneous MPSoCs*; Castrillon and Leupers 2014, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*.

87: Eker, Janneck, et al. 2003, *Taming Heterogeneity—the Ptolemy Approach*; Ptolemaeus 2014, *System Design, Modeling, and Simulation using Ptolemy II*.

88: Eker and Janneck 2003, *CAL Language Report: Specification of the CAL Actor Language*.

89: Stephens 1997, *A Survey of Stream Processing*; Hirzel, Baudart, et al. 2018, *Stream Processing Languages in the Big Data Era*.

90: Thies, Karczmarek, and Amarasinghe 2002, *StreamIt: A Language for Streaming Applications*.

91: Pop and Cohen 2013, *OpenStream: Expressiveness and Data-Flow Compilation of Openmp Streaming Programs*.

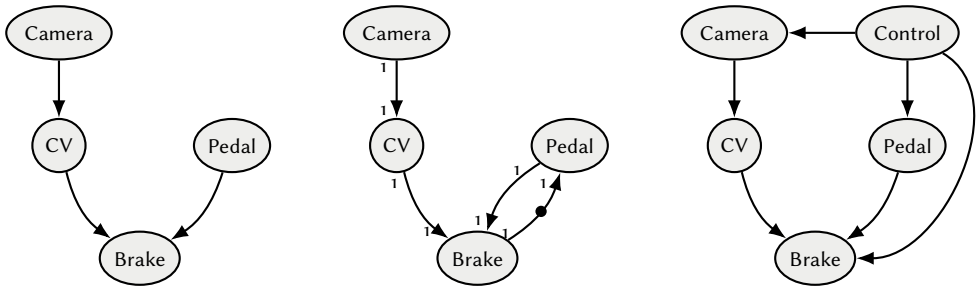
92: Hirzel, Andrade, et al. 2009, *SPL Stream Processing Language Specification*; Hirzel, S. Schneider, and Gedik 2017, *SPL: an Extensible Language for Distributed Stream Processing*.

93: Castrillon, Desnos, et al. 2023, *Dataflow Models of Computation for Programming Heterogeneous Multicores*.

94: Ertel 2019, *Towards Implicit Parallel Programming for Systems*; Suchert et al. 2023, *ConDRust: Scalable Deterministic Concurrency from Verifiable Rust Programs*.

2.4.4 Limitations

The deterministic semantics of KPN and SDF, as well as the clear interfaces and separation of concerns, address several of the CPS design challenges discussed in Section 1.2. Process networks and dataflow models meet the requirements of stream processing applications as they are needed, for instance, in computer vision pipelines (e.g., in autonomous cars) or for audio processing. Due to their ability to exploit data-level and pipeline parallelism, dataflow models are known to perform well in such scenarios. However, regarding CPS design, there are two major limitations. First, the models do not include a notion of time, and second, KPN and SDF have limited expressiveness with respect to reactive behavior.



(a) Hewitt actors **(b)** SDF **(c)** KPN

Figure 2.12: Implementations of a simple emergency brake assistant using different actor-based MoCs.

Consider the application in Figure 2.12a, which denotes a simplified emergency brake assistant (EBA) application as it could be found in a car. This example uses Hewitt actors. There is a camera actor that sends video frames captured by a camera mounted in front of the car. The camera produces a new frame every 20 ms. The frames are processed by a computer vision actor (CDV), which detects obstacles on the vehicles path and estimates their distance. If there is the potential for a collision, the computer vision component sends a message to the brake actor, which actuates the brake. There is also a brake pedal actor, which sends a message to the brake when the driver presses the brake pedal down.

There are several challenges to realizing this application in SDF. For instance, the camera actor is supposed to send a frame every 20 ms, but there is no mechanism to express timed behavior in SDF. Thus, the behavior of the camera actor needs to explicitly implement a waiting mechanism.

In the above description, the computer vision actor only sends a message to the brake if the brake needs to be applied. This, however, is not possible in SDF due to the fixed rates. Thus, the computer vision actor needs to send a token to the brake for every camera frame. Based on the value, the brake actor decides whether to apply the brake or not. This introduces a significant overhead as additional (null) messages have to be sent, and the brake actor fires most often without doing anything.

Incorporating the brake pedal is even more challenging. The channel from the brake pedal to the brake actor implies that for every token received from the computer vision actor, the brake also needs to receive a token from the brake pedal in order to fire. Thus, the brake pedal actor would need to sample the brake pedal sensor every 20 ms and implement a similar waiting mechanism as the camera. Alternatively, we can insert a back edge that allows the brake to poll the brake pedal for every camera frame. This solution is shown in Figure 2.12b. The black dot on the back edge denotes an initial token (also called a *delay*). Without this initial token, the application would immediately deadlock.

In the SDF solution in Figure 2.12b, the polling of the pedal is synchronous to the camera frames and therefore limited to a sampling interval of 20 ms. This is a fundamental limitation of SDF. In CSDF, we could decrease the sampling interval by adjusting the brake's firing rates to only consume a token from the computer vision actor every n firings.

In KPN, we are not limited to fixed firing rates. However, we need to carefully coordinate reading from the channels to avoid deadlocks. A naive KPN implementation of the brake actor is given in Listing 2.2. It reads a token from the computer vision process, applies the brake if needed, then reads from the brake pedal and applies the brake if needed. This implementation

```
while true do
    if readToken(cv) = true then
        applyBrake()
    if readToken(pedal) = true then
        applyBrake()
```

Listing 2.2: Pseudocode of a naive KPN process implementing the brake actor.

implicitly assumes that the brake receives a token from the pedal for every token it receives from the computer vision process. If the brake pedal only sends a message when the brake pedal is pressed, then the brake process in this naive implementation would block until the brake pedal is pressed. Any subsequent tokens that are sent by the computer vision process would be ignored until the brake pedal is pressed.

To avoid potential deadlocks, we need to coordinate the writing to and reading from channels. We can do this implicitly by writing and reading tokens at certain rates, like in SDF or CSDF, or we can do this explicitly by adding a coordinating process. In the solution given in Figure 2.12c, there is an additional control process. It sends tokens to the camera for every frame it should produce, and it sends tokens to the brake pedal to poll its current status. Via an additional channel to the brake, the control process informs the brake at which channel it can next expect a token. Listing 2.3 shows a pseudocode implementation of the brake process. It receives an integer variable from the control process and uses this value to decide from which channels it reads. This solution is more flexible than a CSDF implementation, as it allows in principle to adjust the sampling rates dynamically. However, we are still limited to polling sensors and introduce an overhead for sending unnecessary messages.

```
while true do
    b ← false
    c ← readToken(control)
    if c = 1 then
        b ← readToken(cv)
    else if c = 2 then
        b ← readToken(pedal)
    else if c = 3 then
        b ← readToken(pedal) ∨
        readToken(cv)
    if b = true then
        applyBrake()
```

Listing 2.3: Pseudocode of a KPN process that implements the brake actor and receives instructions from a control actor.

Both in SDF and KPN, it is not possible to react to spontaneous events. If we are willing to sacrifice determinism, then we can implement the example in DDF and simply specify two firing rules for the brake pedal. One reacting to a token from computer vision, and one reacting to a token from the brake pedal. This behavior is commonly called a nondeterministic merge. In this particular example, this solution would be preferable, as we do not require strong consistency and the brake should be applied as quickly as possible.

In summary, KPN and SDF are too restrictive to efficiently express reactive behavior. Moreover, it is not possible to break out of the deterministic semantics in cases where this might be desired. However, as earlier sections argued, nondeterministic models do not provide a suitable alternative for CPS design. Many applications, like the aircraft door example described in Section 2.3.2, require consistency and benefit from the improved testability and reliability of deterministic solutions. Ideally, a MoC for CPS design is reactive and deterministic, but also includes mechanisms for deliberately introducing nondeterminism when this is required for realizing applications like the EBA discussed in this section.

2.5 Models of Time

The MoCs discussed in the previous sections have ignored one crucial aspect of any physical process: *time*. In CPSs, time plays an essential role for two reasons. First, the *cyber* part, i.e., the computation, is realized by physical processes in a computer that take time. Second, the computation needs to interact with the *physical* part, i.e., other physical processes in the environment, and for this interaction to be meaningful, the computation needs a notion of time.[95]

95: E. A. Lee 2009, *Computing Needs Time*.

Time is a subtle concept. Before we discuss in the subsequent sections MoCs that include a notion of time in one form or another, we first introduce in this section various models of time.

2.5.1 Physical Time

Our intuitive understanding of time commonly assumes a Newtonian model. In Newtonian mechanics, time is assumed to be continuous and absolute. The current time is a global state that is instantly shared by all observers. This model of time has proven useful for understanding the dynamics of local and relatively slow-moving systems. However, since Einstein's theories of special and general relativity, we know that Newton's model is imprecise for analyzing the dynamics of fast objects and large distances.

Following the argument given by Lohstroh, E. A. Lee, Edwards, et al., Newtonian time is not a useful model for modern CPSs:[96]

> Today's electronic systems may span the globe, operate with sub-nanosecond timing, and consist of discrete, discontinuous state transitions. These systems are not Newtonian because the order in which physically separated events occur is neither practically knowable nor theoretically well-defined.[97] Distributed systems have no well-defined "current state."

96: Lohstroh, E. A. Lee, Edwards, et al. 2023, *Logical Time for Reactive Software*.

97: Bojowald 2017, *Now: the Physics of Time*; Rovelli 2018, *The Order of Time*.

In this thesis, we adopt the notion of *imperfect readings* of physical time given by Lohstroh, E. A. Lee, Edwards, et al., which assumes that "physical time at a single point in space behaves like a smoothly advancing real number."[96] Let $\mathbb{T}$ denote the totally ordered set of all possible time values that any clock within the system may return. A clock reading the current instance of physical time $\tau \in \mathbb{R}$ returns an imperfect reading $T \in \mathbb{T}$, where neither τ nor the precise mapping of τ to T is knowable. Clocks are not consistent, and different clocks may return different values in $\mathbb{T}$.

2.5.2 Logical Time

Under the laws of relativity, the order of any two events occurring in separate locations may depend on the frame of reference, in which case there is no knowable true order. However, for many applications, e.g., the banking example in Figure 2.6, the correct behavior depends on an unambiguous ordering of events. We can impose such an order by assigning timestamps to events. Lohstroh, E. A. Lee, Edwards, et al. state:[96]

> If two separated components assign to their respective events two timestamps t_1 and t_2 drawn from a totally ordered set $\mathbb{T}$, we can have a clear, unambiguous semantic model of the progression of the system based on the order of these timestamps. This is not a scientific model because we do not demand the ordering of timestamps necessarily match any physical truth, but it is an engineering model that we can implement faithfully.

The notion of timestamps for ordering events logically in a distributed system was first introduced by Leslie Lamport.[98] His notion of timestamps is commonly referred to as Lamport timestamps. These timestamps are not at all related to a wall clock's reading of physical time and are represented by natural numbers ($\mathbb{T} = \mathbb{N}$).

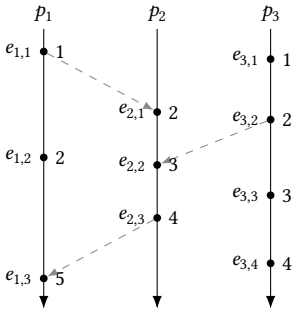


Figure 2.13: Events in three distributed processes annotated with Lamport timestamps.

98: Lamport 1978, *Time, Clocks, and the Ordering of Events in a Distributed System*.

Figure 2.13 shows how Lamport timestamps are assigned to distributed processes that send each other messages. $e_{i,j}$ denotes the jth event of process p_i. Each process keeps a local clock that assigns a timestamp $t_{i,j}$ to the event $e_{i,j}$ such that $t_{i,j}$ is greater than the timestamp of all events causing $e_{i,j}$. The transient binary relation $\rightarrow$ denotes a causal dependency between events. $e_{i,j} \rightarrow e_{k,l}$ if $e_{i,j}$ precedes $e_{k,l}$ in the same process ($i = k$ and $j < l$), or if $e_{i,j}$

sends a message that is received by $e_{k,j}$. The clock is consistent if it fulfills the following property:

$$e_{i,j} \rightarrow e_{k,l} \implies t_{i,j} < t_{k,l}$$

The $<$ relation over timestamps establishes a partial order and is referred to as the *happens before* relation. We can also define a total order by introducing a tie-breaker for events that have the same timestamp (e.g., $e_{1,2}$ and $e_{2,2}$ in Figure 2.13). For instance, a tie-breaker could give events in processes with a lower process identifier precedence over events in processes with a higher process identifier.

The Lamport timestamps do not fulfill the strong consistency property:[99]

$$e_{i,j} \rightarrow e_{k,l} \iff t_{i,j} < t_{k,l}$$

For instance, $e_{1,2}$ in Figure 2.13 has a greater timestamp than $e_{3,1}$, but clearly the events are independent, and $e_{3,1}$ does not cause $e_{1,2}$. Vector timestamps have been proposed as a solution.[100] For n processes, the set of timestamps becomes $\mathbb{T} = \mathbb{N}^n$. Figure 2.14 shows the same processes and events as in Figure 2.13, but with annotated vector timestamps. Using this representation, we can clearly identify if two events are causally related.

Timestamps provide us with a tool for representing the causal relationship of events and creating a partial or total orders of events. However, the timestamps themselves do not provide a semantic notion of how concurrent processes should coordinate their execution. For this, we can build additional coordination protocols that leverage the timestamp mechanism. Lamport provided an example of such a protocol in his original paper.[98] In the following sections, we will discuss MoCs that leverage a similar notion of logical time to coordinate concurrent processes without relying on programmers to manually write such protocols.

To define logical time, we adopt the notation of Lohstroh, E. A. Lee, Edwards, et al. and introduce another abstraction.[101] Let $\mathbb{G}$ denote the totally ordered set of all *tags*. A tag $g \in \mathbb{G}$ denotes a logical time. If two events have the same tag g, then they are *logically simultaneous*. We further require a monotonically increasing function $\mathcal{T} : \mathbb{G} \rightarrow \mathbb{T}$ that maps each tag $g \in \mathbb{G}$ to a timestamp $t \in \mathbb{T}$. In the case of Lamport and vector timestamps, we simply have $\mathbb{G} = \mathbb{T} = \mathbb{N}^n$ (with $n = 1$ in the case of Lamport timestamps), and $\mathcal{T}$ is the identity function.

99: Raynal and Singhal 1996, *Logical Time: Capturing Causality in Distributed Systems*.

100: Fidge 1988, *Timestamps in Message-Passing Systems That Preserve the Partial Ordering*; Friedmann 1988, *Virtual Time and Global States of Distributed Systems*; Schmuck 1988, *The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems*.

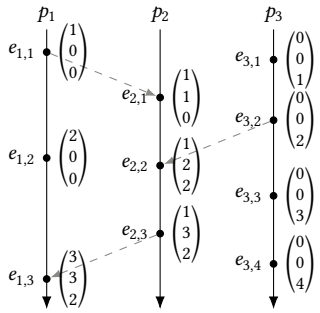


Figure 2.14: Events in three distributed processes annotated with vector timestamps.

101: Lohstroh, E. A. Lee, Edwards, et al. 2023, *Logical Time for Reactive Software*.

2.5.3 Representations of Time

There are many possible choices for $\mathbb{T}$ and $\mathbb{G}$. This subsection discusses a selection of common choices.

For CPSs, it is reasonable to require that timestamps relate to the passage of time as perceived by physical observers. Assuming that physical time is a real number, a reasonable representation for timestamps could be $\mathbb{T} = \mathbb{R}$, or practically, the set of floating-point numbers in a particular representation. However, floating-point arithmetic is subtle, and comparing two floating-point numbers does not always yield the expected result.[102] Therefore, it is typically better to represent time as an integer number.[103]

A common integer representation of time is Unix time, which represents time as the number of seconds (the Unix *epoch*) since 00:00:00 UTC on Thursday, 1 January 1970.[104] Other integer representations can be chosen to increase precision. For instance, the highest precision time representation of the

102: Dawson, Bruce 2012a, *Comparing Floating Point Numbers, 2012 Edition*.

103: Dawson, Bruce 2012b, *Don't Store That in a Float*; Broman et al. 2015, *Requirements for Hybrid Cosimulation Standards*.

104: IEEE Computer Society and The Open Group 2018, *IEEE Standard for Software-Hardware Interface for Multi-Many-Core*.

chrono C++ library is the Unix epoch in nanoseconds represented by a 64-bit integer.[105]

105: cppreference.com 2023a, *Date and Time Utilities*.

Some applications require a more fine-grained mechanism to establish an order between events that have the same timestamp and would otherwise appear logically simultaneous. This is similar to the tie-breaker used for establishing a total order of Lamport timestamps. For this, we can leverage the additional abstraction provided by tags and introduce another dimension to logical time, creating a *superdense time*.[106] Superdense time uses $\mathbb{G} = \mathbb{T} \times \mathbb{N}$, and each tag g is a tuple $g = (t, m)$ consisting of a timestamp t and a superdense time index m that is commonly referred to as the *microstep* of a tag. The function $\mathscr{T}(g) = t$ maps each tag to its timestamp. The total order relation of superdense tags is defined as follows:

106: Maler, Manna, and Pnueli 1992, *From Timed to Hybrid Systems*.

$$(t_1, m_1) < (t_2, m_2) \iff t_1 < t_2 \vee t_1 = t_2 \wedge m_1 < m_2$$

Superdense time allows us to logically order events that are causally related without requiring that their timestamps in $\mathbb{T}$ increase. This is particularly useful in discrete event simulation,[107] which we discuss in the next section.

107: K. H. Kim et al. 1997, *Ordering of Simultaneous Events in Distributed DEVS Simulation*; Rönngren and Liljenstam 1999, *On Event Ordering in Parallel Discrete Event Simulation*.

2.6 Discrete Events

The timed behavior of many physical processes can be modeled using ordinary differential equations and continuous functions. Consider, for example, the CMOS inverter with a capacitive load given in Figure 2.15. This system can be modeled using differential equations, and given the input signal, we can determine a continuous function that models the output signal. For instance, when we send a rectangular pulse to the input, as given in the top-left plot in Figure 2.16, then the output voltage is expected to follow the continuous function given in the plot below.

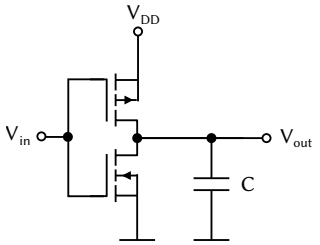


Figure 2.15: A CMOS inverter with a capacity connected to its output.

Circuit engineers commonly use simulation tools to design circuits and analyze a circuit's behavior before manufacturing it. Computers, however, cannot easily handle differential equations and continuous functions directly. Therefore, circuit simulators commonly approximate the solution and use a discrete time axis with a fixed time increment. In such simulators, a solver iteratively predicts the system's state for the next time step based on the current state. The top-right plot in Figure 2.16 shows an approximation for a fixed time step of 5 ns.

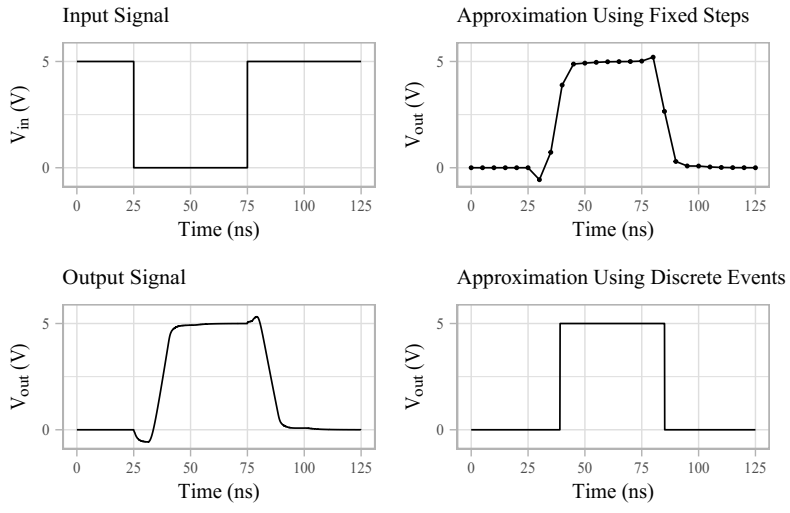


Figure 2.16: An input signal and different models of the output signal of a CMOS inverter.

Such a simulation is typically computationally intensive, especially if the time increments are small for better accuracy and the time span of interest is relatively long. At each time step, the entire set of equations needs to be reevaluated. This is particularly costly if the system state does not change noticeably. Some simulators alleviate this problem by using adaptive time steps, such that updates are computed more frequently if the simulated signal changes quickly. However, we can reduce the computational load of the simulation considerably if we are only interested in certain *event*.

For analyzing digital circuits, it is not commonly required to simulate the output voltage continuously. To understand digital logic, we only need to consider if the voltage level is *low* or *high*—0 or 1. Thus, we can model the system's behavior as a series of discrete events, where an event denotes a change in a signal from low to high or high to low. For example, we can model the input signal as two events $e_{i,l}$ and $e_{i,h}$ with tags $g_{i,l} = 25\,\text{ns}$ and $g_{i,h} = 75\,\text{ns}$.

Assuming that we know the switching delay imposed by the inverter, a simulator can *schedule* new events to model the future changes to the output signal in response to the input events. In this example, the delay is approximately 14 ns for switching from low to high and 10 ns for switching from high to low, and thus the simulator would schedule the output events $e_{o,h}$ and $e_{o,l}$ with tags $g_{o,l} = 39\,\text{ns}$ and $g_{o,h} = 85\,\text{ns}$. The outcome of this "simulation" is shown in the bottom right plot in Figure 2.16.

2.6.1 Discrete Event Simulation

In discrete event simulation (DES),[108] typically a simulation kernel manages the progress of the simulation. It keeps track of the *event queue*, which holds all unprocessed events, provides an application programming interface (API) for scheduling new events, and iterates over the events in the event queue one by one in order of their associated tags. In general-purpose simulation frameworks such as SystemC[109] or SimPy[110], the users can freely define the relevant events and also provide event handlers that are invoked by the kernel when a corresponding event is processed. In simulators that are more specialized for a certain use case, this can be partially automated. For instance, it is possible to automatically derive a DES model from a hardware model written in a hardware description language (HDL) such as Verilog,[111] VHDL,[112] or Chisel.[113]

The discrete events model is quite universal. It can be applied to model a wide range of systems in different domains, and it can be scaled to an arbitrary level of abstraction by redefining the events of interest. In the inverter example, we considered events at the level of the inputs and outputs of individual gates. However, if only the functional behavior of the hardware is of interest and not its timing characteristics, it can be simulated at the register transfer level (RTL). At the RTL level, all events are aligned with the ticks of a set of clocks. Abstracting even further, the interaction between multiple hardware components can be represented as messages instead of individual 1-bit signals. This enables the cycle-accurate simulation of entire processors and multiprocessor systems on a chip (MPSoCs), for instance in SystemC TLM[114] or gem5.[115] In fact, DES is also universal in the sense that different modeling frameworks can be combined.[116]

108: Fishman 2011, *Discrete-event Simulation: Modeling, Programming, and Analysis*.

109: Panda 2001, *SystemC: A Modeling Platform Supporting Multiple Design Abstractions*; Black et al. 2009, *SystemC: From the Ground Up, Second Edition*.

110: Matloff 2008, *Introduction To Discrete-Event Simulation and the Simpy Language*; Team SimPy 2023, *SimPy: Discrete Event Simulation for Python*.

111: D. E. Thomas and Moorby 2008, *The Verilog Hardware Description Language*.

112: Pedroni 2004, *Circuit Design with VHDL*.

113: Bachrach et al. 2012, *Chisel: Constructing Hardware in a Scala Embedded Language*.

114: Maillet-Contoz and Ghenassia 2005, *Transaction Level Modeling*.

115: Binkert et al. 2011, *The gem5 Simulator*; Lowe-Power et al. 2020, *The Gem5 Simulator: Version 20.0+*.

116: Menard, Jung, et al. 2017, *System Simulation with gem5 and SystemC: The Keystone for Full Interoperability*; Blochwitz et al. 2011, *The Functional Mockup Interface for Tool independent Exchange of Simulation Models*.

2.6.2 Limitations

The concept of discrete events is also particularly useful for modeling cyber-physical systems. Since events are tagged, the discrete event model provides a timed semantics, but this notion of time is typically purely logical. Discrete events are most commonly used in simulation, where the tags correspond to estimated times in the prediction model and not to physical readings of time in the real world. The general model is less practical for reasoning about the execution of software in a real system. Also, additional constraints are required for ensuring the deterministic execution of discrete event models. Popular implementations of discrete events like SystemC actually do not guarantee a deterministic execution.[117] However, as we will discuss in the remainder of this thesis, it is possible to use more constrained discrete event models to deterministically coordinate software execution in real systems and to establish a connection between the logical tags of events and physical readings of time.

117: Schumacher et al. 2010, *ParSC: Synchronous Parallel Systemc Simulation on Multi-Core Host Architectures*.

Leveraging discrete events for coordinating the execution of software is also challenging regarding scalability. Since the model assumes a single global event queue and events need to be processed in tag order, it is difficult to leverage parallel hardware and to deal with distributed memories. There exists a wide range of solutions for parallelizing popular discrete event simulation frameworks like SystemC and gem5,[118] but using them remains challenging as they often require manual partitioning and expose users to the pitfalls of threads.

There are also manifold solutions for executing simulations jointly in a distributed system.[119] However, those typically introduce a time quota, which gives a time window in which events are allowed to be executed out of order. This introduces a certain amount of inaccuracy but increases performance as the individual nodes can operate more independently within a certain bound. Such a solution, however, is not suitable for controlling safety-critical systems. More conservative approaches use rollback mechanisms to recover from scenarios where an event was handled too early. But rollbacks are only possible in a simulated environment, not when interacting with the real physical world.

118: Schumacher et al. 2010, *ParSC: Synchronous Parallel Systemc Simulation on Multi-Core Host Architectures*; Chung, J.-K. Kim, and Ryu 2014, *SimParallel: A High Performance Parallel SystemC Simulator Using Hierarchical Multi-threading*; T. Schmidt, G. Liu, and Dömer 2017, *Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation*; Zurstraßen et al. 2023, *par-gem5: Parallelizing gem5's Atomic Mode*.

119: Huang et al. 2008, *Scalably Distributed SystemC Simulation for Embedded Applications*; Cox 2005, *RITSim: Distributed SystemC Simulation*; Alian, D. Kim, and N. S. Kim 2016, *pd-Gem5: Simulation Infrastructure for Parallel/distributed Computer Systems*; Alian, Darbaz, et al. 2017, *dist-gem5: Distributed simulation of computer clusters*.

2.7 Synchronous Languages

The family of synchronous programming languages builds on the *synchronous reactive* MoC,[120] which can be seen as a specialized version of the more general discrete event MoC.[121] The semantics of synchronous languages are defined based on the *synchronous hypothesis*, which makes the assumption that "reactive systems produce their outputs synchronously with their inputs, their reaction taking no observable time."[122] Thus, the synchronous reactive paradigm imposes an idealized view on the real physical behavior of a system. This idealized view enables easy composition of systems without affecting their observable behavior.

120: Halbwachs 1993, *Synchronous Programming of Reactive Systems*.

121: E. A. Lee and Zheng 2007, *Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems*.

122: Benveniste and Gérard Berry 1991, *The Synchronous Approach To Reactive and Real-Time Systems*.

It is considered the compiler's task to deliver an implementation that is faithful to the assumptions made in the synchronous reactive MoC. This idea is rooted in the design process commonly used for digital circuits. Here, the designer is concerned with electrical signals that are exchanged between registers and logical gates to implement a certain function. When considering the functional behavior of the circuit, all logical gates can be considered as producing their outputs synchronously to their inputs, and all events are defined based on the ticks (e.g., rising edge) of a global clock. Only once the

circuit is realized in hardware, we have to ensure that all signals traverse the gates in time before the next tick of the clock.

In the synchronous reactive paradigm, all events are defined with respect to a global clock. This clock is purely logical, and its ticks denote a sequence of instants of computation ($\mathbb{G} = \mathbb{N}$).[123] Since the system is defined as a series of events and synchronous reactions to those events, the synchronous reactive MoC is closely related to discrete event models.[121] However, in contrast to most discrete event systems, which typically require that new events be scheduled with a non-zero delay to ensure causality, the synchronous hypothesis requires that inputs and outputs are synchronous.

123: Lohstroh, E. A. Lee, Edwards, et al. 2023, *Logical Time for Reactive Software*.

Following a denotational semantics approach, synchronous reactive systems are defined as compositions of their components, where at each tick, each component can be modeled as a function that maps the component's inputs to its outputs.[124] Note that this function may change between ticks. Thus, at each tick, the system is defined as a set of equations. Consequently, computation of the system's outputs requires solving the set of equations. In the presence of direct feedback, a unique least fixed point solution needs to be found. We call a synchronous program well-formed, if such a solution can be found in bounded time.[125]

124: Gérard Berry 2000, *The Foundations of Esterel*; Edwards and E. A. Lee 2003, *The Semantics and Execution of a Synchronous Block-Diagram Language*.

125: E. A. Lee and Seshia 2016, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*.

2.7.1 Languages and Tools

The most prominent examples of synchronous languages include Esterel,[126] Signal[127] and Lustre.[128] They all build on the synchronous hypothesis, but their approaches are quite different.[129] While Lustre is a declarative language and primarily describes data flow, Esterel is an imperative language and primarily describes control flow. Lustre is sample driven, which means that it reads inputs and computes updates for each tick of the clock. Esterel, however, is event driven, which means that it computes updates for each input event. While Esterel embraces zero-delay feedback loops and has fixed-point semantics, Lustre requires that programs be acyclic, and any feedback imposes a delay. Signal combines aspects of both approaches. It has declarative sample-driven statements as well as imperative event-driven statements. Signal is designed for specifying open systems as compositions of individual programs, where each program is driven by its own clock. As such, Signal is a *multiclock* language. There is also a multiclock extension for Esterel.[130] Also Lustre allows defining multiple clocks, but only at fractions of the main clock.

126: Boussinot and Simone 1991, *The Esterel Language*; Gérard Berry 2000, *The Foundations of Esterel*.

127: Le Guernic et al. 1991, *Programming Real-Time Applications With Signal*.

128: Halbwachs et al. 1991, *The Synchronous Data Flow Programming Language Lustre*.

129: Benveniste, Caspi, et al. 2003, *The Synchronous Languages 12 Years Later*.

130: Gérard Berry and Sentovich 2001, *Multiclock Esterel*.

Listing 2.4: A Lustre program implementig a stopwatch and a table that represents an execution sequence.

```
1  node stopwatch
2    (tick: bool; reset, start_stop: bool)
3  returns
4    (time: int; running: bool);
5  let
6    time = 0 -> if reset then 0
7                else if running and tick then pre(time)+1
8                else pre(time);
9    running = false -> if start_stop then not pre(running)
10                      else pre(running);
11 tel
```

Instant	0	1	2	3	4	5	6	7
tick	t	t	t	t	t	t	t	t
start_stop	f	f	t	f	f	f	t	f
reset	f	f	f	f	f	t	f	f
time	0	0	1	2	3	0	1	1
running	f	f	t	t	t	t	f	f
pre(time)	?	0	0	1	2	3	0	1
pre(running)	?	f	f	t	t	t	t	f

Listing 2.4 shows an example program in Lustre that implements a simple stopwatch. The program has three inputs `tick`, `start_stop`and `reset`. And it produces the outputs `time`and `running` The table on the right represents one possible execution sequence for this program. The execution proceeds in a sequence of logical time instants. At each instant, the program computes its outputs and state based on its inputs and the previous state. The `pre`

operator, which is used in the example, implements a delay and denotes the value at the previous instant.

The strength of the synchronous reactive paradigm lies in its rigorous mathematical foundation. It allows for a correct-by-construction approach, where the program is a specification of the intended behavior and components can be composed arbitrarily without changing this behavior. Moreover, the safety properties of programs can be formally verified.[131]

These characteristics make synchronous languages particularly attractive for the development of safety-critical systems. Already in the 1980s, Lustre was applied in the development of the Airbus A320, which is the first commercial fly-by-wire aircraft, and in the development of the N4 series of nuclear power plants.[129] This has led to the development of the SCADE language and tool suite.[132] SCADE is a commercial product that is now developed by Ansys, Inc.[133] and has been used in numerous commercial applications, including French nuclear power plants, the Airbus A340-600, and the Hong Kong subway.[129] Also, Esterel was applied in large-scale industry applications in avionics[134] and digital signal processor (DSP) design.[135]

131: Jagadeesan, Puchol, and Von Olnhausen 1995, *Safety Property Verification of Esterel Programs and Applications to Telecommunications Software*; Bouali 1998, *Xeve, an Esterel Verification Environment*; Hagen and Tinelli 2008, *Scaling Up the Formal Verification of Lustre Programs with SMT-Based Techniques*.

132: Abdulla et al. 2006, *Designing Safe, Reliable Systems Using Scade*; Gérard Berry 2007, *SCADE: Synchronous Design and Validation of Embedded Control Software*; Colaço, Pagano, and Pouzet 2017, *SCADE 6: A formal language for embedded critical software development (invited paper)*.

133: Ansys, Inc. 2023, *Ansys SCADE Suite: Model-Based Development Environment for Critical Embedded Software*.

134: Gérard Berry, Bouali, et al. 2000, *Esterel: a Formal Method Applied To Avionic Software Development*.

135: Arditi et al. 1999, *Using Esterel and Formal Methods to Increase the Confidence in the Functional Validation of a Commercial DSP*.

2.7.2 Limitations

While synchronous languages have been used in various safety-critical commercial applications, there are several challenges to applying the approach to a broader range of systems on a larger scale. One such challenge is that Esterel, SCADE, or similar languages are perceived as rather obscure and restrictive by programmers that are used to the sequential code of mainstream programming languages. Moreover, safely integrating large legacy codebases with a synchronous language imposes significant challenges. There are, however, some efforts to make the synchronous semantics more accessible to programmers. The synchronous constructive (SC) MoC, for instance, replaces signals with variables and allows for more programming patterns known from sequential programming while preserving the synchronous semantics.[136] This approach is also implemented in the visual language SCCharts.[137] More recently, HipHop.js integrated Esterel-like language constructs with JavaScript.[138]

Compiling synchronous programs, so that they can efficiently exploit parallel hardware is another challenge. While synchronous programs are inherently concurrent, the resulting parallelism is rather fine-grained. Mapping such fine-grained structures to parallel computation in multiple threads is challenging, as the effort required for synchronizing the threads easily outweighs the benefits gained from parallel execution. Therefore, synchronous languages are typically compiled into single-threaded sequential programs. To efficiently parallelize the execution, the program needs to be partitioned by the compiler[139] or via language-level constructs like futures.[140] Alternatively, specialized hardware can be used to minimize the synchronization overhead.[141]

Moreover, despite the use of logical clocks, synchronous languages do not provide a clear mechanism for handling physical time. In the synchronous reactive paradigm, the progress of physical time is seen as an input to the system. The ticks of a physical clock (e.g., every second) are input events that the system may react to. In the example in Listing 2.4, the stopwatch counts ticks, not physical time. The environment is responsible for providing the ticks of a physical clock at sensible intervals and for choosing how other input events relate to this clock. If outputs should be created with a certain physical delay, this needs to be implemented manually by counting the ticks

136: Hanxleden, Mendler, et al. 2014, *Sequentially Constructive Concurrency-A Conservative Extension of the Synchronous Model of Computation*.

137: Hanxleden, Duderstadt, et al. 2014, *SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications: HW/SW-Synthesis for a Conservative Extension of Synchronous Statecharts*.

138: Gérard Berry and Serrano 2020, *HipHop.js: (A)Synchronous Reactive Web Programming*.

139: Yuan, Yoong, and Roop 2011, *Compiling Esterel for Multi-core Execution*.

140: Cohen, Gérard, and Pouzet 2012, *Programming Parallelism with Futures in Lustre*.

141: Li 2007, *The Kiel Esterel Processor: A Multi-threaded Reactive Processor*; Li and Hanxleden 2012, *Multithreaded Reactive Programming—The Kiel Esterel Processor*.

of the input clock. This requires that the frequency of the input clock be known in advance by the program.

Finally, the synchronous hypothesis only holds as long as the system can keep up with processing the events. This means that the physical time required for computing the response to events may not exceed the interval between subsequent input events. Typically, synchronous languages do not provide mechanisms for detecting or handling such a situation as part of the program.

2.7.3 The Sparse Synchronous Model

The sparse synchronous model (SSM) proposed by Hui and Edwards integrates more tightly with a physical notion of time and also acknowledges that computation may take a significant amount of time.[142] While also following a strictly synchronous semantics, reactions in SSM, may create concurrent tasks that commit their results at a later time instant.[143] The model is sparse in the sense that the underlying clock is fine-grained for high timing precision, but reactions are not processed at every tick of the clock. Computation only occurs in response to input events and when concurrent tasks commit their results. The frequency of the underlying clock is not known by the user, and in contrast to traditional synchronous languages, a change in clock frequency does not change the program's behavior.

142: Hui and Edwards 2022, *The Sparse Synchronous Model on Real Hardware*.

143: This idea is closely related to the concept of asynchronous function calls and futures in Lustre (see 140).

The initial implementation of SSM is C-based but there is also a Lua implementation[144] and a programming language called Scoria.[145] While the SSM is very promising for the development of cyber-physical systems, it currently only provides a rather low-level abstraction and is aimed at deeply embedded devices. Also, the approach cannot yet be extended to coordinate the execution of programs in a distributed system.

144: Hui and Edwards 2023, *Towards Sparse Synchronous Programming in Lua*.

145: Krook et al. 2022, *Creating a Language for Writing Real-Time Applications for the Internet of Things*.

2.8 Logical Execution Time

While the synchronous reactive MoC assumes that inputs and outputs are synchronous, the logical execution time (LET) paradigm takes the opposite approach. The LET model acknowledges that computation takes time and provides a simple mechanism for linking the logical progression of the program to the physical time of the environment.[146]

146: Kirsch and Sokolova 2012, *The Logical Execution Time Paradigm*.

The LET model is motivated by the inherent nondeterminism in classic real-time task models (cf. Section 2.2.4). Consider again the example given in Figure 2.5 and reproduced in Figure 2.17a. Depending on when jobs start their computation and for how long they compute, the concrete data paths may change. Kirsch and Sokolova call this approach bounded execution time (BET), as the classic real-time model provides an upper bound for when jobs write their outputs but does not define the ordering precisely.

In the LET model, however, the time at which jobs write the results is defined precisely on a logical timeline. The set of tags is equal to the set of possible readings of physical time $\mathbb{G} = \mathbb{T}$. Each task has an associated LET, which we denote with L and which is equal to the task's relative deadline. A job that is released at physical time T reads its inputs at tag $g = T$. The job must complete its computation at physical time $T + L$, but may finish earlier. However, even when finishing earlier, the job does not write the results immediately. Instead, the results are written at tag $g + L$, precisely at the deadline of the job. Both reading inputs and writing outputs are logically instantaneous.

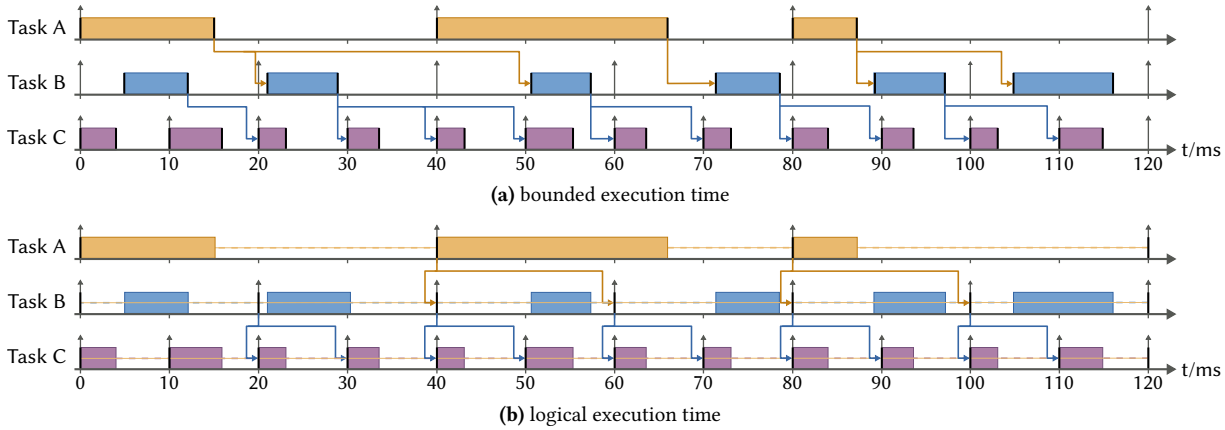


Figure 2.17: An example schedule for three tasks and their read-write-dependencies using a classic real-time model (bounded execution time) and the logical execution time model. This figure is loosely based on Figure 3 in Gemlau et al. 2021, *System-Level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-Time Automotive Software*.

Physical time and logical time are tightly coupled in the LET paradigm. The logical timeline is used to define the ordering of reads and writes between jobs, but logical and physical time are assumed to progress in lockstep. For this reason, the LET of a task may not be less than its WCET.

The principles of LET are adopted in various languages, like Giotto[147] and the Time Definition Language (TDL).[148] Also the time-triggered architecture (TTA) uses a similar time model.[149] On multi-core processors, LET enables deterministic lock-free communication, which makes it attractive for modeling parallel embedded software.[150] Most importantly, LET was adopted in the Classic Platform of the automotive standard AUTOSAR,[151] which the next section will discuss in more detail.

The principles of LET can also be applied to modeling the interaction of tasks running on different processors that communicate over a bus or network. In system-level LET, the communication between processors is also modeled as tasks with an associated LET. The LET of communication tasks is determined by the worst-case network latency and the maximum synchronization error of the physical clocks of each node. This allows us to apply the same principles as for modeling tasks on a single processor to the system level.[152]

LET enables deterministic concurrency for embedded real-time systems. It can be conveniently applied to any program that can be modeled with periodic tasks, like control software. However, LET is quite restrictive and has limited expressiveness, which hinders its application to a wider range of applications that require reactions to sporadic events. Like the synchronous reactive paradigm, which assumes that zero time passes between reading inputs and writing outputs, LET also defines a fixed value of logical time that passes when a task executes. However, as we will explore in this thesis, a more general approach allows the programmer to freely set the logic delay imposed by computation instead of requiring a fixed value. This exposes an interesting trade-off.

In addition, tasks are a rather low-level abstraction and are not convenient for modeling and understanding large systems. Typically, modeling large systems requires some form of component-based design, where components encapsulate state and functionality and can be composed vertically and horizontally, as it is possible in actors and related models. However, there is no hierarchy in task-based models. Also, dependencies between tasks are only described implicitly by the release time and deadlines of jobs and

147: Henzinger, Horowitz, and Kirsch 2003, *Giotto: A Time-Triggered Language for Embedded Programming*.

148: Pree and Templ 2008, *Modeling with the Timing Definition Language (TDL)*.

149: Kopetz and Bauer 2003, *The Time-Triggered Architecture*.

150: Hennig et al. 2016, *Towards Parallelizing Legacy Embedded Control Software Using the LET Programming Paradigm*; Biondi et al. 2017, *Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores*.

151: Ernst, Ahrendts, and Gemlau 2018, *System Level LET: Mastering Cause-Effect Chains in Distributed Systems*; AUTOSAR 2022j, *Specification of Timing Extensions*.

152: Ernst, Ahrendts, and Gemlau 2018, *System Level LET: Mastering Cause-Effect Chains in Distributed Systems*; Gemlau et al. 2021, *System-Level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-Time Automotive Software*; Köhler et al. 2023, *Robust Cause-Effect Chains With Bounded Execution Time and System-Level Logical Execution Time*.

by the values they read and write, but there is no semantic notion of data dependencies in the model.

2.9 CPS Frameworks and Standards

While the previous sections introduced a range of MoCs, this section investigates practical approaches that are used in industry for CPS design. Several industry standards and software frameworks have evolved to facilitate the joint development of CPSs. Typically, such frameworks provide a unified abstraction layer on top of the underlying hardware and the operating system (OS). And they also provide a unified communication mechanism that allows multiple components to interact and exchange data. In principle, this foundation enables the composition out of complex CPSs out of smaller components that are developed independently and are potentially provided by different vendors.

Ideally, the basis for such a framework should be a well-chosen MoC that addresses the challenges discussed in Section 1.2. However, as we will see in this section, this is not commonly the case. Without such a solid foundation, however, a framework does not provide application designers with the appropriate tools for actually managing the fundamental challenges in CPS design.

This section discusses two popular CPS frameworks in more detail: the AUTomotive Open System ARchitecture (AUTOSAR) and the Robot Operating System (ROS). Parts of the introduction to AUTOSAR were published before in Menard, Goens, Lohstroh, et al. 2020, *Achieving Determinism in Adaptive AUTOSAR*.

2.9.1 AUTOSAR

AUTOSAR is a global development partnership of automotive companies and other interested parties.[153] The automotive industry also faces the challenges discussed in Section 1.2. In particular, stringent safety and real-time requirements, but also a dramatic increase in the complexity and computational demands of automotive software. AUTOSAR aims to address these challenges by standardizing the design process, the runtime environment, and the common software framework.

153: AUTOSAR 2023, *AUTomotiv Open System ARchitecture*.

The partnership maintains two standards called *Classic Platform (CP)*[154] and *Adaptive Platform (AP)*[155] that serve different goals and requirements, as well as the *Foundation* that is shared between the two platforms. The Classic Platform is widely established in industry and mostly intended for hard real-time applications with low computational complexity deployed on ECUs with single-core or simple multicore processors. The CP standard facilitates a task-based programming model and also includes support for the LET paradigm.[156]

154: AUTOSAR 2022d, *Methodology for Classic Platform*.

155: AUTOSAR 2022c, *Methodology for Adaptive Platform*.

156: AUTOSAR 2022j, *Specification of Timing Extensions*.

The Adaptive Platform was introduced more recently to handle applications with high computational demands, facilitate an ongoing interaction with a changing environment, and allow systems to adapt to such changes. AUTOSAR AP is a service-oriented architecture that is based on a POSIX-compliant operating system. The software stack includes a middleware that handles communication between services as well as the Runtime Environment for Adaptive Applications (ARA), which provides common APIs and services.[157] While the standard does not specify the precise middleware and

157: AUTOSAR 2022a, *Explanation of Adaptive Platform Design*.

also supports third-party solutions, AUTOSAR suggests using the Scalable Service-Oriented Middleware over IP (SOME/IP) protocol.[158]

158: AUTOSAR 2022e, *SOME/IP Protocol Specification*; AUTOSAR 2022f, *SOME/IP Service Discovery Protocol Specification*.

The remainder of this section introduces and discusses the Adaptive Platform in more detail.

Service Interfaces

AP applications consist of one or multiple software components (SWCs) that communicate via services that they may provide or request. We call an SWC that provides a service a *server*, and an SWC that requests a service a *client*. Client and server roles may be fulfilled by the same SWC. SWCs provide or request services as needed; the binding between clients and servers is determined at runtime by the middleware through service discovery.[159] The dynamic binding of services is the core mechanism for providing adaptivity in AP.

159: AUTOSAR 2022g, *Specification of Communication Management*.

The service interfaces are fully specified at design time and are composed of methods, events, and fields. While events are one-way messages that the server initiates and the client handles, methods are two-way messages that the client initiates and the server responds to. Fields are state variables exposed by the server. Each field may provide a get method, a set method and an event that indicates state changes.

Listing 2.5 shows the pseudocode definition of an example interface for an accumulator service. The interface provides a method for initializing the service with an initial integer value, a method for adding an integer value, and a method for getting the current accumulated value. The interface further defines an event that the service notifies whenever its internal accumulated value changes.

```
interface AccumulateService
    method init(x: i32)
    method add(x: i32)
    method get(): i32
    event valueChanged: i32
```

Listing 2.5: An example interface for an accumulator service.

Service Communication

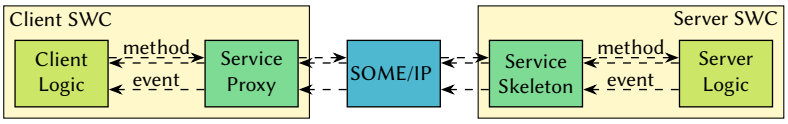


Figure 2.18: Communication mechanism in AUTOSAR AP. Client and server use auto-generated proxies and skeletons to communicate with their peers.

SWCs abstract over the precise middleware by using *proxies* and *skeletons* that are generated from a service description.[160] Figure 2.18 illustrates the overall communication mechanisms in AUTOSAR AP. A skeleton is an abstract interface that a server needs to implement in order to provide a service. A proxy is an object that a client receives when requesting a service.

160: AUTOSAR 2022b, *Explanation of ara::com API*.

Client and server communicate via the proxy and skeleton objects. For instance, the client can invoke a method on the proxy object, which automatically sends a message via the middleware to the server. It also returns a future as a placeholder for the server's response. The server translates the incoming message back into a method call on the skeleton interface. The implementation for this method, which is provided by the server, is expected to return a future. As soon as the server fulfills the corresponding promise, the skeleton sends a message back to the client.

Applications in AUTOSAR AP commonly consist of multiple SWCs. Each individual SWC can be considered a full program as it is mapped to a process on the target platform during deployment. While the service-oriented communication model of AUTOSAR AP specifies how SWCs interact, it does not

specify how SWCs should be implemented. The standard, however, suggests a thread-based coding style.

AP SWCs are implemented in C++. Listing 2.6 provides a simplified example implementation for a client that interacts with the accumulator service defined in Listing 2.5. The client instantiates a proxy object on line 2 to interact with the accumulator service. The construction of the proxy blocks until a connection with a server providing the service is established by the middleware. The server needs to be implemented and started separately. Next, the client initializes the accumulate service with 1, adds 2, obtains the result and, finally, prints the result.

```
1 int main() {
2   AccumulateServiceProxy s{};
3   s.init(1);
4   s.add(2);
5   auto result = s.get();
6   std::cout << result.get();
7   return 0;
8 }
```

Listing 2.6: A client program using the accumulate service to add 1 and 2.

Nondeterminism

The SoA paradigm utilized by AUTOSAR AP is similar to the Hewitt actor model in that it describes a system as a set of components that exchange messages. SoA specializes Hewitt actors by providing a notion of interfaces and defining the precise types of expected messages that components may send or receive when interacting with a service. The execution strategy, however, is similar to the Hewitt actor model. Consequently, SoAs and specifically AUTOSAR AP inherit the nondeterminism of Hewitt actors. In addition, the use of RPCs for implementing service method calls masks the interaction with concurrent components.

Consider again the client code in Listing 2.6. At first glance, with C++ being a procedural language, the code suggests that the program prints a value of 3. However, potentially unbeknownst to the programmer who wrote the client program, the proxy object implements the methods `init` and `add` as non-blocking RPCs. And while we can assume that the server implementation enforces mutual exclusion between the execution of method invocations to avoid data races on its internal state, by default, the runtime environment maps each invocation to a different thread,[161] meaning the order in which the calls are handled is determined purely by the thread scheduler. Consequently, no order is enforced by the server on the handling of calls to `init`, `add`, and `get`, which leads to nondeterministic results.

161: AUTOSAR 2022h, *Specification of Execution Management*.

Depending on the precise interleaving of method calls, we may observe one out of four possible results. By running the program repeatedly and recording the printed result, we can plot the distribution of possible results, as shown in Figure 2.19. This assumes that the internal value of the accumulation service is initialized to 0 by default. Clearly, such a program is not very useful.

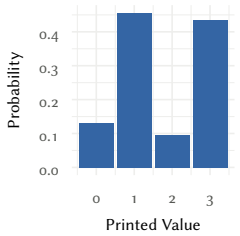


Figure 2.19: Distribution of possible results for the client program in Listing 2.6.

Of course, the client could serialize each method call by waiting for the future returned by the server to resolve before invoking the next method call. Listing 2.7 shows an updated version of the client program that invokes `get()` on the returned future to actually retrieve the returned value. In the case of `init` and `add`, no value is returned, but `get()` blocks nonetheless until the server finishes processing the method. In addition, the server could instruct the runtime to use a single thread rather than multiple for execution. However, multi-threading may be necessary to meet performance requirements, yet it is often far from obvious how this may lead to nondeterminism in realistic AUTOSAR applications, which are, of course, incomparably more complex than this simple example.

```
1 int main() {
2   AccumulateServiceProxy s{};
3   s.init(1).get();
4   s.add(2).get();
5   auto result = s.get();
6   std::cout << result.get();
7   return 0;
8 }
```

Listing 2.7: Corrected client program using blocking calls to wait on the returned future objects.

As this thesis argues, the software designer should not be responsible for engineering solutions to concurrency problems to achieve determinism. Rather, the underlying model should allow for the exploitation of concurrency in ways that preserve determinism, making it easy to write deterministic programs and requiring explicit directions from the programmer to forgo

determinism. In AUTOSAR AP, however, the gained adaptivity and flexibility compared to CP come at the cost of inherent nondeterminism; despite the goal of supporting safety-critical and possibly autonomous applications.

We can identify three distinct sources of nondeterminism in AUTOSAR AP:

1. The suggested programming model for the implementation of individual SWCs is based on threads. As Section 2.1 discusses, threads make it notoriously difficult to engineer deterministic concurrent software. AUTOSAR AP provides coding guidelines to avoid the problems with threads, but nondeterminism is still likely to creep in, especially when code evolves over time.[162]
2. The order in which SWCs process incoming messages is undefined. If two clients call the same method, the server may process them in either order. This is analogous to the problems with nondeterminism in the actor MoC (cf. Section 2.3.2).
3. Point-to-point in-order message delivery could be achieved by the middleware and underlying TCP/IP network stack, but this is not a formal requirement in AUTOSAR AP.

162: Gu et al. 2015, *What Change History Tells Us about Thread Synchronization*.

One provision for deterministic execution that AUTOSAR AP introduces is the so-called "deterministic client", which provides a task-based programming model for the implementation of SWCs.[163] However, its scope is limited to *individual* SWCs and does not consider communication between SWCs. Therefore, this solution only addresses the first source of nondeterminism. Applications that consist of multiple communicating deterministic clients can still exhibit nondeterminism via the second and third source of nondeterminism.

163: AUTOSAR 2022h, *Specification of Execution Management*.

More recently, Bellassai et al. explored how the LET paradigm could be applied to AUTOSAR AP.[164] While this should alleviate the problems with nondeterminism in AUTOSAR AP, this approach inherits the limitations of LET discussed in Section 2.8.

164: Bellassai et al. 2023, *Supporting Logical Execution Time in Multi-Core Posix Systems*.

2.9.2 ROS

Robot Operating System (ROS) is an open-source framework that facilitates the development of robotic applications.[165] It is similar to AUTOSAR AP in its scope and overall approach. ROS applications are divided into components called *nodes*. ROS provides a hardware abstraction layer and libraries that bundle common functionality. This allows ROS nodes to execute on a wide range of hardware platforms. Despite its name, ROS is not an operating system and assumes to run on top of a Unix-like OS, possibly with real-time capabilities.

165: Quigley et al. 2009, *ROS: an Open-source Robot Operating System*; Koubaa 2016, *Robot Operating System (ROS)—The Complete Reference (Volume 1)*.

Similar to AUTOSAR AP, ROS relies on a middleware to orchestrate the communication between nodes. While the initial version of ROS provided a custom middleware, ROS 2 instead builds on the DDS standard.[166] DDS implements a publish/subscribe protocol. Consequently, ROS nodes communicate via topics that they may publish or subscribe to.

166: Macenski et al. 2022, *Robot Operating System 2: Design, Architecture, and Uses in the Wild*; OMG 2015, *Data Distribution Service*.

Also similar to AUTOSAR AP, ROS inherits the problems of its underlying models. Concurrency within a node is expressed using plain threads or tasks, and the interaction of multiple nodes via publish/subscribe is nondeterministic (cf. Sections 2.3.2 and 2.3.3). In a reasonably complex system, this can easily lead to situations where components are in an inconsistent state,[167]

167: Bateni, Lohstroh, et al. 2023, *Risk and Mitigation of Nondeterminism in Distributed Cyber-Physical Systems*.

which could have fatal consequences. While the ROS framework and middleware provide a lot of flexibility, the application designers are responsible for building reliable distributed software without the appropriate tools.

2.10 Discussion and Conclusion

This chapter surveyed a wide range of MoCs, all of which are actively being used for CPS design, both in research and industry. However, the discussion of problems and limitations also showed that all existing models have their pitfalls. Commonly, a MoC is well-suited only for a particular use case, e.g., for safety-critical systems with limited complexity or for large-scale distributed systems with only light safety requirements.

Reconsider the challenges discussed in Section 1.2. We inferred that the ideal MoC for CPS design should be *concurrent*, *deterministic*, *reactive*, *scalable*, and *timed*. All the MoCs discussed in this chapter allow for expressing concurrent computation. Figure 2.20 summarizes the findings of this chapter by characterizing each of the surveyed MoCs regarding the remaining four desired properties.

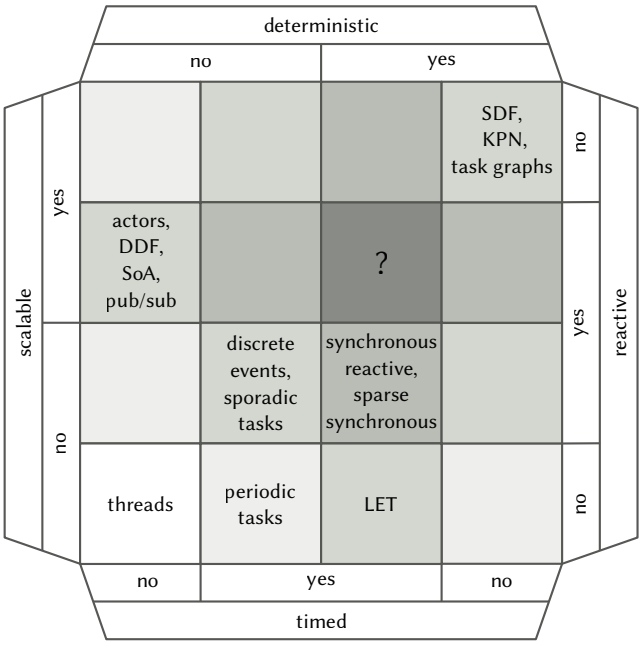


Figure 2.20: Overview of MoCs and frameworks for cyber-physical systems. (Repeated from Figure 1.2)

2.10.1 Discussion

Obviously, threads are a particularly bad choice. They are not deterministic, do not offer a notion of time, cannot easily be scaled to distributed systems, and by themselves do not offer support for reactive behavior. Yet, modern CPS frameworks like ROS and AUTOSAR AP assume a thread-based programming model for implementing nodes and SWCs. While we can make an effort to "fix" threads, e.g., by appealing to the programmer's discipline and by providing libraries that add a notion of time or support for reactive programming patterns, a MoC that does not need to be fixed would serve as a much more solid foundation for CPS design.

Hewitt actors alleviate the pitfalls of threads regarding low-level data races, and the message passing paradigm allows for scaling transparently from

local execution with a few parallel cores up to massively parallel distributed systems. Hewitt actors are reactive in nature and have been proven effective in practice for modeling large-scale concurrent software. However, they have not been designed for CPS use cases. Hewitt actors neither provide a notion of time nor can they guarantee determinism. The same applies to related models like SoA, publish/subscribe, and DDF.

SDF, KPN, and task graphs are closely related to Hewitt actors but use more restrictive rules on how actors communicate and execute behavior. These restrictions allow for guaranteeing a deterministic execution, but come at the expense of reactivity. Such models are particularly useful for developing streaming applications, e.g., the processing of video frames, but are not generally applicable to CPS design due to their lack of reactivity. Also, dataflow models do not expose a notion of time.

In the category of timed models, various forms of real-time task models are the most prominent representatives. Real-time task models have a long history in safety-critical and time-sensitive applications. However, tasks are closely related to threads, and two tasks that share access to a common resource are subject to low-level data races. Hence, the task model is nondeterministic. Periodic task models commonly use a static schedule, but sporadic task models are more reactive in nature. There are various variants that allow for expressing adaptivity or parallelism within tasks. However, the model cannot easily scale to distributed execution.

LET provides a deterministic variant of the general task model. LET tasks orchestrate reads and writes on a logical timeline. This makes the execution deterministic, as the data paths solely depend on this logical ordering. Although it has been demonstrated that the model can be applied to distributed execution, Figure 2.20 characterizes LET as non-scalable. This is because the task model does not provide a convenient abstraction for composing large-scale systems. There is no notion of components or hierarchy that supports organizing an application. Also, the LET model is commonly limited to periodic tasks, which makes it non-reactive.

Finally, the synchronous reactive model also has a strong notion of logical time at its core. It assumes that computation is logically instantaneous and is both deterministic and reactive. However, the paradigm cannot easily be applied to general-purpose languages. Typically, behavior is defined on a fine-grained level, which hinders scalability. Most compilers for synchronous languages generate sequential programs, and exploiting parallel hardware or even enabling distributed execution is challenging. The sparse synchronous model integrates a physical notion of time and allows for expressing computation that is not logically instantaneous. In principle, this could allow for a more scalable approach, but this hasn't been demonstrated yet.

The background color of each cell in Figure 2.20 indicates how many of the properties *concurrent*, *deterministic*, *reactive*, *scalable*, and *timed* are fulfilled in this position. It is evident that the dark gray box is empty, and no model provides all the desired properties. Moreover, most of the neighboring cells, which fulfill three out of four properties, are empty. Only the synchronous reactive and sparse synchronous models come close to providing the ideal model for CPS design, but they are challenging to scale up and do not provide much flexibility.

The overview of concurrent MoCs given in this chapter is far from complete. The literature on concurrent MoCs is rich and describes more models than can possibly be described in a single chapter. Therefore, this chapter primarily focuses on models that are commonly used for expressing concurrent executable programs, in particular in the domain of CPS design.

There are also a range of popular but more abstract models that are often used for reasoning about certain aspects of concurrent computation but less commonly for describing actual executable behavior. This includes Statecharts,[168] Petri nets,[169] and various process calculi like the π-calculus[170] and communicating sequential processes.[171]

168: Harel 1987, *Statecharts: a Visual Formalism for Complex Systems*.

169: Petri 1962, *Kommunikation mit Automaten*; Peterson 1977, *Petri Nets*.

170: Milner 1999, *Communicating and Mobile Systems: The Pi Calculus*.

171: Brookes, Hoare, and Roscoe 1984, *A Theory of Communicating Sequential Processes*; Hoare 1985, *Communicating Sequential Processes*.

2.10.2 Conclusion

The discussion in this chapter and the summary in Figure 2.20 highlight a semantic gap. Since there is no model that combines all properties, application designers need to carefully choose a MoC based on the requirements of a specific use case. If the application needs to be reactive and scaled to a relatively large system, then currently the only suitable option are the Hewitt actor model and similar models. This is why AUTOSAR AP and ROS adopt the SoA and publish/subscribe paradigms. However, this comes at the cost of introducing nondeterminism and, hence, compromising predictability, testability, and ultimately safety.

This thesis argues that the gap can be bridged by utilizing a novel MoC that combines aspects of various MoCs discussed in this chapter. The next chapter introduces such a new MoC.

Reactors: Deterministic Actors in Adaptive AUTOSAR

3

The reactor model is a novel MoC proposed by Lohstroh, Romeo, et al.[1] It combines various aspects of other MoCs to fill the gap identified in Sections 1.3 and 2.10. Intuitively, we can describe reactors as deterministic actors with a discrete event execution semantics and explicitly declared ports and connections.[2] A logical timeline is used to order events and ensure a deterministic execution.

The reactor model is reactive, ensures deterministic execution, has a well-defined notion of time, and can be scaled to a high number of parallel threads or executed on distributed systems. This chapter introduces the reactor model in detail (Section 3.1) and gives several examples to provide an intuitive understanding of how reactor programs execute (Section 3.2). Chapter 4 provides more insight on the timed semantics of reactors, and Chapter 5 focuses on scalability.

This chapter further introduces a full implementation of the reactor model in C++ and presents the DEAR framework, which integrates the reactor model with AUTOSAR AP (Section 3.3). Based on this integration, Section 3.4 analyzes an industrial use case implemented in AUTOSAR AP, illustrates how the application exposes nondeterminism, and shows how a reactor implantation can be realized to achieve a deterministic implementation.

1: Lohstroh, Romeo, et al. 2019, *Reactors: A Deterministic Model for Composable Reactive Systems*; Lohstroh 2020, *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*.

2: Lohstroh, Schoeberl, et al. 2019, *Actors Revisited for Time-Critical Systems*; Lohstroh and E. A. Lee 2019a, *Deterministic Actors*.

3.1 The Reactor Model

This section provides an informal introduction to the reactor model and discusses examples based on a *visual* syntax.[3] Chapter 4 further introduces a *textual* syntax for reactor programs. For a formal introduction of the reactor model, the interested reader may refer to Lohstroh, Romeo, et al.[4] Rossel et al. also provide a formalization of the core reactor semantics in Lean and a proof of determinism.[5]

The reactor model builds on the principles of some well-established paradigms (cf. Chapter 2). In particular, it combines the notion of concurrency and reactive execution semantics of Hewitt actors with explicitly declared connections, similar to the ones found in dataflow models, a clear notion of simultaneity, as it is found in synchronous languages, and a logical notion of time, similar to those found in discrete event modeling. Time is a first-order citizen in the reactor model. This includes a well-defined notion of *logical* and *physical* time as well as relations between the two.

Some examples, figures, and arguments presented in this section were published before in Menard, Lohstroh, et al. 2023, *High-Performance Deterministic Concurrency Using Lingua Franca*.

3: Hanxleden, E. A. Lee, et al. 2022, *Pragmatics Twelve Years Later: A Report on Lingua Franca*.

4: Lohstroh, Romeo, et al. 2019, *Reactors: A Deterministic Model for Composable Reactive Systems*; Lohstroh 2020, *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*.

5: Rossel et al. 2023, *Provable Determinism for Software in Cyber-Physical Systems*.

3.1.1 Reactor Elements

Similar to actors in the actor model, a reactor is the universal building block in the reactor model. Each reactor is defined by the composition of its contained elements. Figure 3.1 gives a legend of the visual representation for reactor elements that we use throughout the thesis, and an example reactor is shown in Figure 3.2. The following lists and discusses all reactor elements:

- *Reactors* may contain other reactors.
- *Reactions* implement the behavior of a reactor. They are executable building blocks that are comparable to functions in general-purpose programming languages. However, unlike functions, reactions cannot be called directly. Instead, they are *triggered* by certain events and, thus, react to these events.
- *State variables* allow a reactor to encapsulate state (similar to Hewitt actors or classes in object-oriented languages). Reactions have full read and write access to all state variables in their reactor.
- *Timers* are triggers that produce events at fixed, periodic intervals.
- *Logical actions* are triggers that can be *scheduled* by reactions to create new future events.
- *Physical actions* are similar to logical actions, but they may be used to schedule events asynchronously from a context outside the reactor program. This is a key mechanism for handling nondeterministic inputs to the program.
- *Startup and shutdown triggers* are special event sources. Startup produces an event when the program starts, and shutdown produces an event right before the program terminates.
- *Ports* are a reactor's interface to the surrounding context. A reactor may send events to other reactors by setting its output ports, and it may receive events from other reactors via its input ports.
- *Connections* connect one upstream port to one or multiple downstream ports within the scope of the reactor containing the connection.
- *Dependencies and antidependencies* declare the *triggers*, *sources* and potential *effect* of reactions. Triggers are all event sources (i.e., ports, timers, and actions) that trigger the execution of the reaction. Sources are additional ports and actions that a reaction may read from when executing, but that do not trigger the reaction. Both sources and triggers are dependencies of the reaction. The set of antidependencies includes all ports and actions that a reaction may have an effect on (i.e., by setting a port or scheduling an action).
- *Mutations* are similar to reactions, but they are allowed to modify the containing reactor and its elements. While mutations are part of the reactor model, at the moment of this writing, there is no complete implementation of mutations.

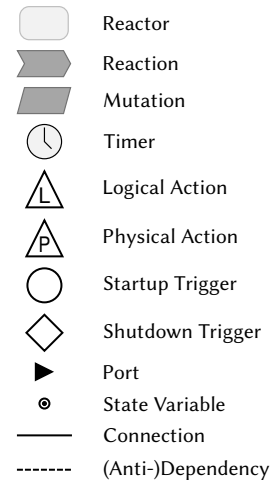


Figure 3.1: Visual representation of reactor components.

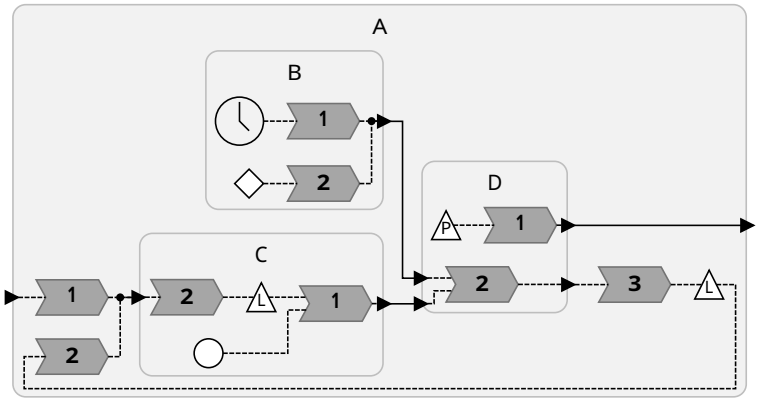


Figure 3.2: An example reactor.

3.1.2 Logical and Physical Time

While, in principle, other choices are possible, we will assume that reactors use a super-dense time and define the set of tags as $\mathbb{G} = \mathbb{N}^2$. The set of timestamps is the set of natural numbers representing the number of nanoseconds since the Unix epoch $\mathbb{T} = \mathbb{N}$.

All events have an associated tag $g \in \mathbb{G}$ that is used to order events on a logical timeline. Events with identical tags are logically simultaneous. Similar to synchronous and functional-reactive languages, all computation is logically instantaneous, i.e., no logical time passes while a reaction executes. Thus, all events that a reaction produces on its output ports have the same tag as the events that triggered the reaction. Relaying events via connections is also logically instantaneous.

The initial tag g_0 is determined when the program starts. It is set to $g_0 = (T_0, 0)$, where $T_0 \in \mathbb{T}$ is the current reading of physical time when the program starts. All startup triggers produce one event with the tag g_0.

Timers are defined by a period $p \in \mathbb{N}$ and an offset $o \in \mathbb{N}$. They automatically produce a series of events with tags $g_{t,n} = (\mathscr{T}(g_0) + o + p \cdot n, 0)\ \forall n \in \mathbb{N}$. If the interval is zero, then only one event at tag $g_t = (\mathscr{T}(g_0) + o, 0)$ is produced.

Actions provide a mechanism for reactions to schedule new events with a tag strictly greater than the tag at which the reaction is triggered. Logical actions schedule events with a delay $d \in \mathbb{N}$ relative to the current tag $g = (t, m)$. The tag of the new event is defined by the tag delay function $\mathscr{D} : \mathbb{G} \times \mathbb{N} \to \mathbb{G}$ that is given by

$$\mathscr{D}(t, m, d) = \begin{cases} (t, m+1) & \text{for } d = 0 \\ (t+d, 0) & \text{for } d > 0\,. \end{cases}$$

Since a delay of 0 increments the microstep of the tag, it is commonly referred to as a *microstep delay*.

Physical actions schedule events with a delay $d \in \mathbb{N}$ relative to the current reading of physical time $T \in \mathbb{T}$. The tag of the resulting event is defined by the function $\mathscr{P} : \mathbb{T} \times \mathbb{N} \to \mathbb{G}$ that is given by:

$$\mathscr{P}(T, d) = \begin{cases} (T, 0) & \text{for } d = 0 \\ \mathscr{D}(T, 0, d) & \text{for } d > 0\,. \end{cases}$$

Since the assignment of tags to events created via physical actions, does not depend on the current state of the reactor program, i.e., the current tag, they can be scheduled asynchronously from contexts outside of the reactor program, like an interrupt handler or an external thread. The assignment of tags via physical actions is nondeterministic, but once tags are assigned, events are processed in tag order.

Section 4.4 discusses logical and physical time and the relations between the two timelines in more detail.

3.1.3 Concurrency and Parallelism

The use of statically declared ports, connections, and reaction dependencies distinguishes reactors from more dynamic models like actors or other asynchronous message-passing frameworks where communication is purely based on addresses. The fixed topology of reactors provides two key advantages. First, it achieves a separation of concerns between the functionality of components and their composition. Second, it makes explicit at the interface level what dependencies exist between components. As a consequence, a dependency graph can be derived for any composition of reactors.

Reactions denote concurrent computations. The dependency graph organizes reactions into a partial order that captures all scheduling constraints that must be observed to ensure that the execution of concurrent reactions yields

deterministic results. Because this graph is valid irrespective of the contents of the code that executes when reactions are triggered, reactions can be treated as black boxes.

The dependency graph is an acyclic precedence graph (APG). It defines precisely in what order logically simultaneous reactions need to be executed. That is, if any two reactions are enabled at a particular tag and there exists a path in the APG from one to the other, then they must execute in the order given by the path. If no such path exists, then the reactions are independent and may be executed in parallel without introducing data races or deadlocks. This gives a precise definition of when reactions may be executed in parallel without breaking determinism.

Figure 3.3 shows the dependency graph for the example reactor given in Figure 3.2. The solid arrows represent dependencies that arise because one reaction (possibly) sends data to the other via ports and connections. The dashed arrows represent dependencies that arise because the two reactions belong to the same reactor. Analogous to the behaviors of actors, reactions in the same reactor are mutually exclusive. Each reaction has an associated priority that is indicated by the numbers in the reaction labels in Figure 3.2. The priorities define a total order for the execution of reactions within one reactor.

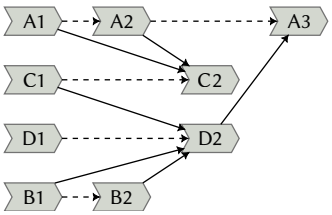


Figure 3.3: Dependency graph of the example reactor in Figure 3.2.

The dependency graph in Figure 3.3 indicates that reactions 1, 2 and 3 of reactor A (labeled "A1", "A2", and "A3") need to be executed in sequence because they are contained in the same reactor. The graph also shows that reaction 2 of reactor D needs to be executed after reaction 2 of reactor B due to a data dependency. However, the graph also denotes which reactions are independent and may be executed in parallel. For instance, reaction 1 of reactor D and reaction 1 of reactor C are independent, and reaction 3 of reactor A is independent of all reactions in C.

Note that actions do not imply a dependency and are thus not represented in the APG. This is because scheduling an action creates a future event with a tag strictly greater than the current tag. This logical delay captures the causal relation between the reaction that schedules an event and the reaction that is triggered by the event. The dependency graph, however, only defines the causal relation of logically simultaneous reactions (i.e., within the same tag). For this reason, the dependency graph needs to be acyclic, as otherwise there would be no well-defined causality. Any dependency cycles in reactor programs can be resolved by introducing a logical action, and thus a logical delay, to break one of the dependencies and moving part of the computation to a later tag.

3.1.4 Execution

The execution of reactor programs is governed by a runtime. Most importantly, the runtime includes a scheduler that keeps track of all scheduled future events, controls the advancement of logical time, and invokes any triggered reactions in the order specified by the dependency graph while also exploiting parallelism where possible. Lohstroh, Romeo, et al. sketched a scheduling algorithm for reactor programs,[6] which was later refined.[7] This section presents a variation of this basic algorithm that is adapted to the notation used in this thesis and simplified in some aspects. Section 5.2 discusses an optimized version of this algorithm.

Figure 3.4 gives a high-level overview of the principle scheduling mechanism. The runtime keeps track of the current logical time and manages three queues: the event queue, which stores all future events; the reaction queue, which

6: Lohstroh, Romeo, et al. 2019, *Reactors: A Deterministic Model for Composable Reactive Systems*, pp. 72, 75.

7: Lohstroh 2020, *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*, pp. 33–35, 45–49.

stores reactions that are triggered at the current tag; and the ready queue, which holds all reactions that are ready for execution. Reactions within the ready queue are picked up for execution by a pool of parallel worker threads. The runtime provides two procedures, SET and SCHEDULE, that an executing reaction may use to trigger additional reactions by setting a port or to create future events by scheduling an action.

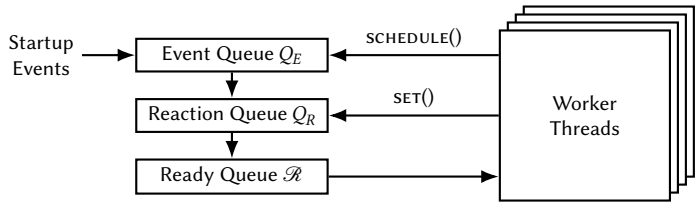


Figure 3.4: Overview of the general scheduling mechanism for reactors.

Listing 3.1 shows the pseudocode implementation of SET. The procedure uses a second procedure called RECURSIVESET to recursively walk along the outgoing connections. For each port that is visited, the procedure writes the value to the port and collects the set of all reactions that have declared this port a trigger. Finally, SET inserts the combined set of all triggered reactions into the reaction queue. A global mutex is used to protect against data races in the queue. Since setting ports and relaying messages is logically instantaneous, the reactions are triggered immediately, and the event queue is left unchanged.

```
1:  procedure SET(port, value)
2:      ▷ Recursively set all ports and obtain the set of all triggered reactions. ◁
3:      R ← RECURSIVESET(port, value)
4:      LOCK(mutex)                          ▷ Lock mutex to avoid data races.
5:      Q_R ← Q_R ∪ R        ▷ Insert triggered reactions into the reaction queue.
6:      UNLOCK(mutex)
7:
8:  procedure RECURSIVESET(port, value)
9:      WRITEVALUE(port, value)          ▷ Write the actual value to the port.
10:     ▷ Obtain all reactions that the port triggers. ◁
11:     R ← GETTRIGGEREDREACTIONS(port)
12:     ▷ Recursively walk the connections and call RECURSIVESET on all
          downstream ports. ◁
13:     for all p ∈ GETCONNECTEDDOWNSTREAMPORTS(port) do
14:         R ← R ∪ RECURSIVESET(p)
15:     return R
```

Listing 3.1: Pseudocode implementation of the SET procedure that may be used by reactions to set the value of an output port and trigger downstream reactions.

The SCHEDULE procedure shown in Listing 3.2 allows reactions to schedule future events via actions. The procedure first locks the global mutex to prevent concurrent accesses to the data structures and then determines the tag at which the new event is created, as discussed in Section 3.1.2. Finally, the new event is created and inserted into the event queue. If the event queue already contains an event at the given tag for the given action, then the old event is replaced by the new one.[8]

8: An implementation of the reactor model might choose other conflict resolution strategies, such as deferring the new event to a later tag. However, we will assume that events are overwritten in this thesis.

The main task of the scheduler is to control the advancement of logical time and to decide when to execute triggered reactions. The NEXT procedure shown in Listing 3.3 implements the core of the scheduler's functionality. It advances logical time to the tag of the earliest events in the event queue and then processes all reactions triggered by those events as well as all subsequent reactions that may be triggered via ports.

The NEXT procedure first locks the global mutex to protect against concurrent accesses to the event queue. While no reaction is executing at this point,

a concurrent process could still try to schedule a physical action. After obtaining the mutex, the procedure peeks into the event queue to determine the next tag g_{next}. The scheduler will only proceed with executing this event if $T \geq \mathscr{T}(g_{\text{next}})$. This is if the current reading of physical time is greater than or equal to the timestamp of the next tag. Otherwise, the scheduler will wait until physical time has advanced far enough or the event queue is modified by scheduling a physical action from an external context.

We refer to this waiting mechanism as a physical time barrier. It effectively assigns a physical meaning to logical tags, and it allows for the precise scheduling of events with respect to the physical timeline, which is important if reactions have side effects or control actuators that are interacting with the physical world.

On line 12, the current tag is advanced to the next tag, and then the procedure begins with processing the events at the current tag. First, she scheduler clears all values on ports and actions that might have been set during the previous iteration and initializes the reaction queue as well as the set of executing reactions. Then the scheduler extracts all events at the current tag from the event queue (lines 15 and 16) and finally releases the mutex.

Before starting the execution of reactions, the procedure iterates over all events in lines 18 to 22 to initialize the value of all actions that trigger at the current tag and to insert all reactions that are triggered directly by these events into the reaction queue. The execution of reactions starts once the procedure enters the while loop in between lines 24 and 47.

Within the loop, the procedure first cleans up the set of executing reactions and removes any reactions that are done (lines 25 to 28). In the first iteration of the while loop, there are no execution reactions yet, and hence nothing is cleaned up. If there are worker threads available, the scheduler then determines the set of all ready reactions (the ready queue in Figure 3.4). That is the set of all reactions for which all dependencies are met as indicated by the APG. If there are ready reactions, then one is selected and sent to a worker thread for execution (lines 34 to 37).

If there are no ready reactions, then all reactions that are in the reaction queue have unmet dependencies. Therefore, the scheduler waits until one of the worker threads completes executing a reaction (line 40) before trying again. Also, if there are no threads available for executing a reaction, the scheduler waits until one becomes idle (line 43). Finally, the scheduler also needs to wait for worker threads to finish if there are no more reactions in the reaction queue and there are still executing reactions (line 47).

```
1:  procedure SCHEDULE(action, value, delay)
2:      LOCK(mutex)
3:      ▷ Determine the tag of the new event. ◁
4:      if ISPHYSICAL(action) then
5:          T ← GETPHYSICALTIME()
6:          g_e ← (T, 0)
7:      else
8:          g_e ← 𝒟(g, delay)        ▷ Calculate tag relative to the current tag g.
9:      ▷ Create a new event for the action with the given value and tag. ◁
10:     e ← CREATEEVENT(action, value, g_e)
11:     ▷ If there is already an event scheduled at tat g, then remove it from
            the queue. ◁
12:     Q_E ← Q_E ∖ {e′ ∈ Q_E | action = GETACTION(e′) ∧ g_e = GETTAG(e′)}
13:     Q_E ← Q_E ∪ {e}                  ▷ Enqueue the new event.
14:     UNLOCK(mutex)
```

Listing 3.2: Pseudocode implementation of the SCHEDULE procedure that may be used by reactions to schedule future events on an action.

```
1:  procedure NEXT()
2:      LOCK(mutex)
3:      while true do                                  ▷ Synchronize with physical time.
4:          T ← GETPHYSICALTIME()
5:          g_next ← GETTAG(PEEK(Q_E))
6:          if T ≥ 𝒯(g_next) then
7:              break
8:          else
9:              UNLOCK(mutex)
10:             WAITUNTILTIMEOREVENTQUEUECHANGE(𝒯(g))
11:             LOCK(mutex)
12:     g ← g_next                                     ▷ Advance logical time.
13:     CLEARALL()                                     ▷ Clear all ports and actions.
14:     Q_R, E ← ∅, ∅          ▷ Initialize Q_R and the set of executing reactions E.
15:     ℰ ← {e ∈ Q_E | GETTAG(e) = g}                  ▷ Extract all events at the current tag.
16:     Q_E ← Q_E \ ℰ
17:     UNLOCK(mutex)
18:     for all e ∈ ℰ do
19:         ▷ Set the value of the action triggered by the event e.                ◁
20:         WRITEVALUE(GETACTION(e), GETVALUE(e))
21:         ▷ Add all triggered reactions to the reaction queue.                   ◁
22:         Q_R ← Q_R ∪ GETTRIGGEREDREACTIONS(GETACTION(e))
23:     ▷ Execute all reactions triggered at the current tag.                      ◁
24:     while Q_R ∪ E ≠ ∅ do
25:         ▷ Remove all done reactions from E.                                    ◁
26:         for all r ∈ E do
27:             if ISDONE(r) then
28:                 E ← E \ {r}
29:         if Q_R ≠ ∅ then
30:             if THREADISAVAILABLE() then
31:                 ▷ Obtain the set of all reactions that are ready for execution. That is, they are in Q_R and they do not have a dependency on any other reaction in Q_R or any reaction in E that is currently executing.                                                ◁
32:                 ℛ ← GETREADYREACTIONS(Q_R, E)
33:                 if ℛ ≠ ∅ then
34:                     ▷ Select one reaction and execute it in one of the available worker threads.                                                    ◁
35:                     r ← SELECT(R)
36:                     E ← E ∪ {r}; Q_R ← Q_R \ {n}
37:                     EXECUTEINTHREAD(r)
38:                 else
39:                     ▷ All reactions in Q_R depend on at least one executing reaction. We wait until a thread finishes executing a reaction before we try again.                                                ◁
40:                     WAITUNTILTHREADBECOMESIDLE()
41:             else
42:                 ▷ There is no worker thread available and we wait until one becomes available.                                                    ◁
43:                 WAITUNTILTHREADBECOMESIDLE()
44:         else
45:             if E ≠ ∅ then
46:                 ▷ There are no more reactions to execute, but we have to wait until all executing reactions finish their execution and both Q_R and E are empty.                                                ◁
47:                 WAITUNTILTHREADBECOMESIDLE()
```

Listing 3.3: The main scheduling procedure that advances logical time to the next tag, and then processes all events and triggered reactions at this tag.

Since the reactor model has a discrete event semantics, this scheduling algorithm is closely related to the main event loop used in discrete event simulators like gem5 and SystemC. However, in contrast to most discrete event simulators, the reactor model provides detailed information about dependencies. Leveraging this information, it becomes possible to exploit parallelism without sacrificing determinism or accuracy.

3.2 Example Reactor Programs

To provide a better understanding of how reactor programs operate and how the reactor model helps to solve the problems of other related models discussed in Chapter 2, this section discusses selected example programs.

3.2.1 Bank Account

Consider again the bank account examples in Figure 2.6 on Page 16 that illustrate the problem of nondeterminism in actor programs. Using reactors, we can implement these examples deterministically.

Figure 3.5 shows a reactor implementation of the deposit/withdrawal actor example in Figure 2.6a. The program consists of three reactors: `UserA`, `UserB`, and `Account`. Both users have an output port that is connected to the respective input port at `Account`, allowing the users to send requests to the account. Both users have a reaction that is triggered by a timer. The timer of `UserA` is configured to produce an event with a tag 1 s after program startup; the timer of `UserB` is configured to trigger an event 2 s after program startup. The corresponding reactions simply send a deposit or withdrawal request by setting the user's output port. The yellow boxes in Figure 3.5 are an annotation to indicate the reaction behavior, and they do not have a semantic meaning.

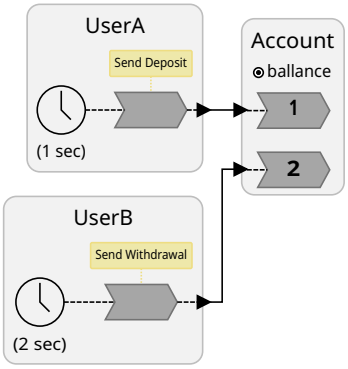


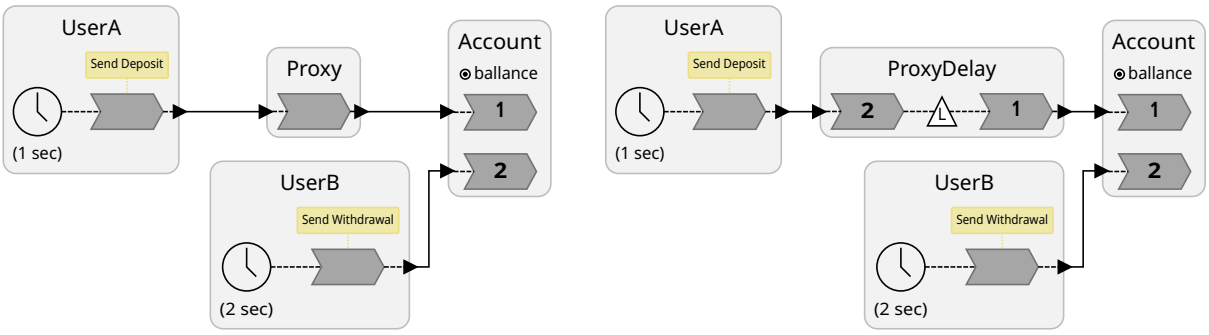
Figure 3.5: Reactor implementation of the account deposit and withdrawal actor program Figure 2.6a introduced in Section 2.3.

The `Account` reactor defines two reactions, one for each of its inputs. Both reactions will simply try to apply the requested change to the balance, which is stored in a state variable. We will assume that the balance is initialized to zero and that `UserA` sends a deposit message of 20 credits while `UserB` sends a withdrawal message of 10 credits.

When executed, the program will wait for 1 s before triggering the timer of `UserA` and invoking its reaction. The event produced by this reaction will trigger reaction 1 in `Account`, which is invoked immediately after the first reaction completes. 2 s after program startup, `UserB` reacts and subsequently triggers reaction 2 of `Account`. In this example, the deposit event (+20) occurs earlier than the withdrawal event (−10), and hence the reactor execution semantics ensures that the account processes the deposit event before the withdrawal event, meaning the balance will not become negative.

Since reactions and communication via ports are logically instantaneous, we can also introduce a proxy reactor, as shown in Figure 3.6a, without changing the order in which events are processed. Even when we reconfigure the timers to trigger logically simultaneously, meaning that both reactions in Account are triggered at the same tag, the resulting program will be deterministic as the APG ensures that reactions are executed in a well-defined order.

To deliberately change the order in which events occur, a delay can be introduced using a logical action, as shown in Figure 3.6b. Upon receiving an input, reaction 2 of `ProxyDelay` is triggered, which schedules the logical action with a delay. This creates a new event that, when processed, triggers



(a) Adding a proxy reactor.

(b) Adding a proxy introducing a logical delay.

Figure 3.6: Variants of the deposit and withdrawal example in Figure 3.5 that use an additional proxy reactor.

reaction 1 of the `ProxyDelay` reactor, which retrieves the original value from the action and forwards it to the output port. If we assume that a delay of 1 s is used for scheduling the logical action, then the deposit message from `UserA` will only arrive at `Account` 3 s after startup. Hence, the deposit message will be processed *after* the withdrawal message from `UserB`, causing B's request to be denied.

In the examples discussed above, we have hard-coded the order in which the users send requests by using timers and, thus, assigned fixed tags to the request events. While using a predefined order is useful for testing and demonstration, reactor programs that are deployed in practice need to be able to handle sporadic asynchronous inputs. Concretely, in the account example, we need to handle asynchronous events that are created when physical users initiate withdrawal or deposit requests.

We can use physical actions as shown in Figure 3.7 to model asynchronous inputs. Since physical actions assign tags based on the current reading of physical time, the order between events scheduled by physical actions is nondeterministic in the sense that it is not defined by the program. However, once those tags are assigned, for example, to deposit or withdrawal requests by a user, the processing of the events is deterministic and occurs in tag order. Hence, the tags assigned to externally initiated events are considered as part of the *input*, and given this input, the program remains deterministic. This approach draws a clear perimeter around the deterministic and therefore testable program logic while allowing it to interact with sporadic external inputs.

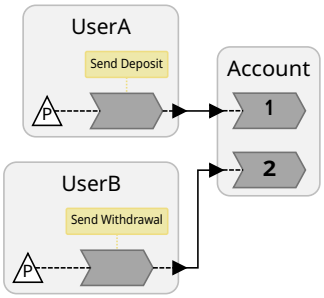


Figure 3.7: Variant of the program in Figure 3.5 that uses physical actions to model user inputs.

3.2.2 Brake Assistant

The previous example showed how reactors enable the deterministic execution of concurrent programs. While the reactor paradigm imposes some restrictions compared to more general MoCs like Hewitt actors, it is more general than the other deterministic models that Chapter 2 discusses. Consider again the brake assistant example that is shown in Figure 2.12 on Page 23 and discussed in Section 2.4.4. Compared to deterministic dataflow and process network MoCs, expressing timed and reactive behavior is natural using reactors.

Figure 3.8 shows a reactor implementation of the brake assistant example. `Camera` simply uses a timer to trigger events every 20 ms and send frames to the `ComputerVision` reactor in reaction to these events. If `ComputerVision` detects an obstacle, it sends a message to the `Brake` reactor. Unlike in SDF, it is not obligatory for a reaction to send a message on each of its outputs.

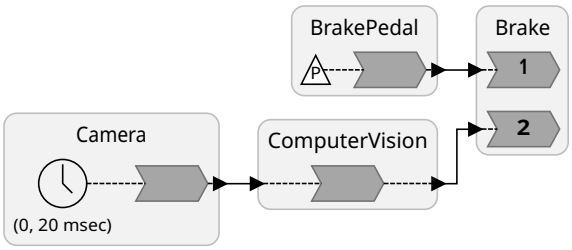


Figure 3.8: Reactor implementation of a simplified brake assistant as discussed in Section 2.4.4.

The reaction may or may not set any declared output ports. Downstream reactions will only be triggered if the port is actually set.

The brake pedal is modeled using a physical action that is scheduled when the driver hits the pedal. The reaction in `BrakePedal` simply sets the output port when the physical action triggers. `Brake` implements two reactions, one for each input port, but both actuate the physical brake as soon as they are executed.

In contrast to more restrictive deterministic models, the reactor model allows for easily expressing both regular behavior in fixed intervals (i.e., the processing of camera frames) and reactions to sporadic inputs (i.e., the brake pedal). However, there are some subtleties that need to be considered when reasoning about the execution of this program, which we will discuss in Chapter 6.

3.3 Integrating Reactors with Adaptive AUTOSAR

In order to demonstrate the usefulness of the reactor model for industrial CPS use-cases, this section introduces a first C++ implementation of the reactor model called DEAR.[9] The DEAR framework implements a C++ reactor runtime and provides integrations for AUTOSAR AP. The framework is designed to achieve deterministic execution of distributed programs in AP. This section introduces the DEAR framework in more detail.

9: Menard 2023, *DEAR: Discrete Events for Adaptive AUTOSAR*.

The DEAR framework consists of a C++ implementation of the reactor model, as well as libraries for translating between reactor communication and the service interfaces of AP. The framework provides type-safe mechanisms for the definition of reactors with ports, actions and reactions. This also includes mechanisms for composing reactors to form deterministic programs, as well as a simple implementation of the scheduling algorithm discussed in Section 3.1.4.[10]

10: The initial implementation of the reactor model in the DEAR framework was significantly extended and optimized later. It became a standalone library called *reactor-cpp*, which is discussed in more depth in Sections 4.7.1 and 5.2.

The reactor implementation establishes a foundation for the design and execution of individual deterministic SWCs. However, in order to build useful programs, we also need to establish a mechanism for deterministic communication between SWCs and for coordinating execution. The discussion in the remainder of this section focuses on this coordination aspect.

This section presents material that was previously published in Menard, Goens, Lohstroh, et al. 2020, *Achieving Determinism in Adaptive AUTOSAR*.

3.3.1 Transactors

Service interfaces are the main abstraction used for communication in AUTOSAR AP.[11] Recall Figure 2.18 on Page 35, which visualizes the communication mechanism. Server and Client communicate through service skeletons

11: AUTOSAR 2022g, *Specification of Communication Management*.

and proxies that are code-generated. The actual communication is implemented transparently on top of a middleware like SOME/IP.

To enable composition of deterministic applications from deterministic SWCs, we need a mechanism for transporting tagged messages. This is challenging since the standard for AUTOSAR AP explicitly specifies the interface that SWCs use for communication.[12] Exposing reactor ports directly to the interface of SWCs would break compatibility with the standard. We can work around this by introducing transactors that translate between the service-oriented interfaces of SWCs and the event-based input and output ports of reactors.

12: AUTOSAR 2022b, *Explanation of ara::com API*.

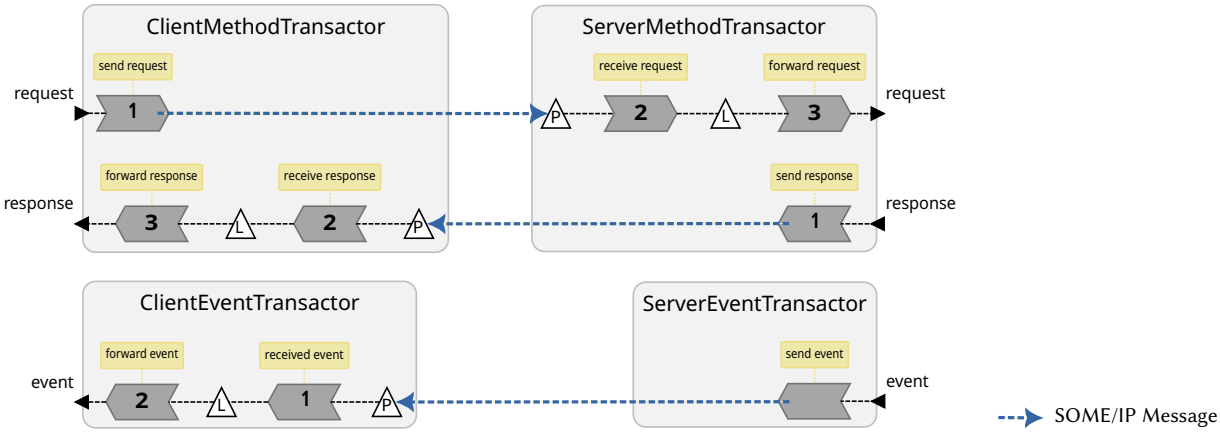


Figure 3.9: The transactors implemented in the DEAR framework provide a reactor interface for interacting with the events and methods of AUTOSAR services.

The DEAR framework provides four distinct transactors that are shown in Figure 3.9. Each transactor is implemented as a reactor and enables other reactors that interact with the transactor via ports and connections to send or receive messages through regular AUTOSAR service interfaces.

The *client method transactor* allows clients to interact with a given service method. It holds a reference to a service proxy object. When receiving a message on the request port, the proxy method transactor forwards this message via SOME/IP to the service by calling the corresponding method on the proxy object. When the service returns the response, an internal callback schedules the physical action. The reaction that is triggered by the physical action determines the correct tag at which the response should be inserted into the local event queue. It then schedules the logical action precisely at this tag. Finally, the reaction triggered by the logical action forwards the response to the response port. From there on, other reactors that implement the client logic can handle the response.

Similarly, the *server method transactor* allows servers to interact with a given service method. It holds a reference to an implementation of the service skeleton. When a client invokes a method, the server receives a request via SOME/IP, which is handled by a callback in the skeleton. Analogous to responses in the proxy method transactor, this callback schedules the physical action. Then the request is scheduled at the correct tag and finally forwarded via the request port to a reactor implementing the server logic. When the external reactor returns the response on the response port, the triggered reaction forwards the response to the client via SOME/IP using the skeleton object.

A similar pair of transactors exists for service events.[13] Events are a one-way communication mechanism. They allow a server to send data to clients.

13: This refers to the concept of events in AUTOSAR service interfaces, which is not to be confused with events in reactors or discrete events in general.

The transactor implementation for events is equivalent to the handling of responses in the method transactors.

Since service fields in AP are composed of a get-method, a set-method and an event, interaction with fields requires the use of one event and two method transactors. Given a service interface, the transactors required for interacting via this particular interface can be automatically generated. This establishes a well-defined reactor interface for any given service interface.

3.3.2 Distributed Execution

Since SWCs in AUTOSAR AP are implemented as separate processes that might run on distributed computers connected over a network, a mechanism for coordinating the distributed execution of multiple reactor programs is required. For this, we can leverage the *safe-to-process* analysis known from Programming Temporally Integrated Distributed Embedded Systems (PTIDES)[14] or Google Spanner.[15]

The PTIDES approach makes three key assumptions. First, it assumes that we know the WCET w for each unit of computation. In the case of the reactor model, these are the reaction bodies. Second, PTIDES assumes that distributed components have synchronized physical clocks with a bounded clock synchronization error c, which is the case in AP.[16] Finally, it assumes that network communication has a bounded latency l.

Based on these assumptions, we can infer that if a reaction sends a message at tag $g = (t, m)$, then this message will arrive at the receiver no later than the physical time point $T = t + w + l + c$ according to the receiver's physical clock. After the time point T, no message with a timestamp greater than t can arrive at the receiver if the assumptions are correct. Consequently, a message that arrives later than at $t + w + l + c$ indicates a violation of the assumptions. Such a violation, however, can be easily detected at runtime and handled by the application in compliance with its safety requirements.

To preserve determinism and the reactor semantics, the runtime needs to make sure that all events are processed in tag order. Before processing an event with tag g, the runtime needs to determine that no unprocessed event with a tag less than g exists or can still arrive. In local execution, this is trivial as there is only a single event queue to be checked. However, in distributed execution there are multiple event queues, and a node has no direct access to the queues in other nodes. Therefore, we need to carefully consider when it is safe to process an event. Based on PTIDES, we can infer that at physical time $T = t + w + l + c$ no event with a timestamp less than t can arrive. Thus, at physical time $t + w + l + c$ it is safe to process any events with a tag up to $g = (t, m)$.

The methodology of PTIDES also considers logical delays and the precise dependencies between individual components. Using PTIDES, we can calculate precise safe-to-process offsets for any point in the system. However, the DEAR framework only considers a special case of PTIDES. If we require that sending a message over the network also imposes a logical delay d of precisely $d = w + l + c$, then any event with tag $g = (t, 0)$ becomes safe to process at physical time $T \geq t - d + w + l + c = t$. Therefore, with this additional constraint, it becomes sufficient to rely on the fact that the reactor runtime will not process events before the current physical time exceeds the timestamp of the event.

Figure 3.10 visualizes both the logical and physical delays imposed by sending a message over the network in DEAR. In this special case of PTIDES, the

14: Y. Zhao, J. Liu, and E. A. Lee 2007, *A Programming Model for Time-Synchronized Distributed Real-Time Systems*; Derler, Feng, et al. 2008, *PTIDES: A Programming Model for Distributed Real-Time Embedded Systems*.

15: Corbett et al. 2013, *Spanner: Google's Globally Distributed Database*.

16: AUTOSAR 2022i, *Specification of Time Synchronization*.

safe-to-process offset is always zero, and the logical delay accounts for the worst-case physical delay between sender and receiver. This approach is equivalent to the one taken by system-level LET.[17] However, as Section 4.5 discusses, there are more general approaches to coordinating distributed reactor execution.

17: Ernst, Ahrendts, and Gemlau 2018, *System Level LET: Mastering Cause-Effect Chains in Distributed Systems*; Gemlau et al. 2021, *System-Level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-Time Automotive Software*.

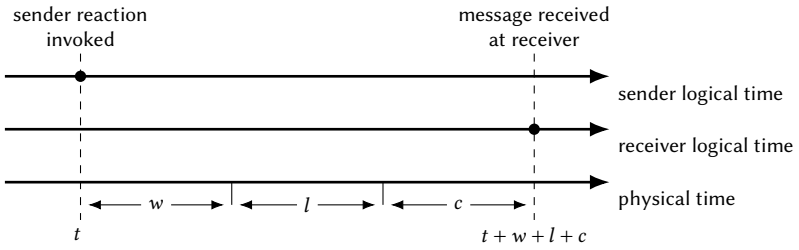


Figure 3.10: The logical and physical delay imposed by sending a message over the network in DEAR.

3.3.3 Implementation

Implementing reactor communication on top of AUTOSAR AP requires a mechanism for exchanging tagged messages between SWCs. However, implementing such a mechanism is challenging as AP does not provide any support for associating metadata like tags with service method invocations or event notifications. This, however, is required for the transmission of tagged messages between SWCs.

DEAR includes a minor modification of the library that binds to the SOME/IP middleware. This modification introduces a bypass that can be used optionally to append tags to outgoing messages and to retrieve tags from incoming messages if available. This modification is not in violation of the standard. It can be seen as the introduction of a new third-party middleware that extends over SOME/IP by allowing the transmission of tagged messages. While the transactors use the regular AUTOSAR AP service proxies and skeletons, for each event notification or method call, they store the corresponding tag such that it can be picked up by the modified SOME/IP middleware before transmitting the payload over the network.

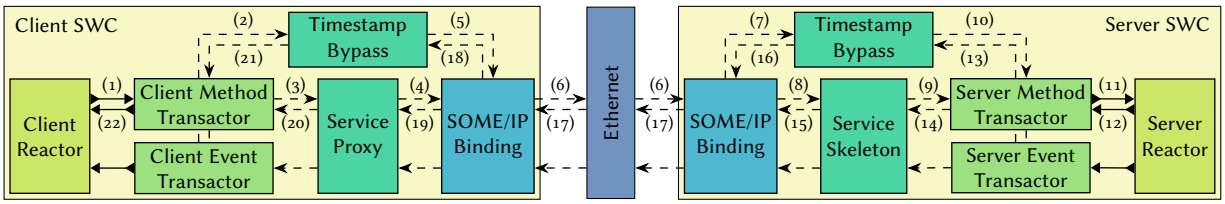


Figure 3.11: Integration of reactors in AUTOSAR AP using the DEAR framework. Special reactors (transactors) translate between the reactor implementation of the SWC logic and the service interface that the SWC exposes to its environment.

The entire process of transmitting tagged messages between SWCs in DEAR is illustrated in Figure 3.11. The sequence starts with a client that invokes a service method call on a remote service. In the reactor implementation, this corresponds to producing an event with tag $g_C = (t_C, 0)$ on the output port connected to the request input port of the client method transactor (1).[18] The client method transactor has a configurable delay $d_C = w_C$ that compensates for the WCET of the sending reaction. A reaction of the transactor handles the request by sending the timestamp $t_C + d_C$ to the timestamp bypass (2) and invoking the actual method call on the service proxy object (3). Thereby, it forwards the data associated with the incoming event as method arguments. The service proxy calls the SOME/IP binding (4) to prepare a network message. The modified binding retrieves $t_C + d_C$ from the timestamp bypass (5)

18: For the sake of simplicity we will assume that all tags have a microstep of zero. In principle, also non-zero microsteps can be handled by this mechanism.

and attaches it to the SOME/IP message, which it then sends over the network to the server (6).

Upon receiving the network message on the server side, the modified SOME/IP binding retrieves $t_C + d_C$ from the message and sends it to the local timestamp bypass (7) before invoking the corresponding method call on the service skeleton (8). This invokes a callback of the server method transactor (9), which schedules a physical action to notify the reactor runtime of the external event (cf. Figure 3.9). The reaction triggered by this physical action retrieves $t_C + d_C$ from the timestamp bypass (10) and schedules the logical action with tag $(t_C + d_C + l + c, 0)$, accounting for the worst-case network latency l as well as the maximum clock skew c between platforms. The reaction to this action produces an event on the output port that forwards the method arguments to the reactor that implements the server logic (11). The server logic then reacts to this event.

The server eventually sends a response by producing an event with the tag $g_S = (t_S, 0)$ with $t_S \geq t_C + d_C + l + c$ on the output port connected to the input port of the service method transactor (12). The transactor has a configurable delay d_S that compensates for the server's WCET w_S. The reaction in the transactor that processes the response sends the timestamp $t_S + d_S$ to the timestamp bypass (13) and returns the data associated with the event to the service skeleton (14). The service skeleton calls the SOME/IP binding (15) to create a response message. The binding retrieves $t_S + d_S$ from the timestamp bypass (16) and attaches it to the outgoing message, which it then sends over the network to the client (17).

The client SOME/IP binding retrieves $t_S + d_S$ from the message and sends it to the timestamp bypass (18) while forwarding the return value to the service proxy (19). This invokes a callback in the client method transactor (20), which schedules the internal physical action. The reaction triggered by this physical action retrieves $t_S + d_S$ from the timestamp bypass (21) and schedules the logical action with tag $(t_S + d_S + l + c, 0)$ to account again for transmission latency and clock synchronization error. Finally, the reaction to this action produces an event on the output port of the transactor (22).

While the DEAR framework enables transparent, deterministic composition of reactor-based SWCs, it also supports composing reactor-based SWCs with regular service implementations that do not communicate via tagged messages. The default behavior of the DEAR transactors is to fail when receiving messages without an associated timestamp, but they can also be configured to tag received messages with the physical time at which they were received. This approach treats the arrival of untagged messages the same way reactors deal with the arrival of sporadic sensor readings. This essentially provides backward compatibility with existing service implementations and adds the ability to gradually introduce reactor-based SWCs into an existing code base.

3.4 Case Study: The Adaptive Platform Demonstrator

Building on the DEAR framework, this section analyzes a concrete automotive application and demonstrates how reactors can be utilized to achieve determinism in AUTOSAR AP. The results presented in this section were published before in Menard, Goens, Lohstroh, et al. 2020, *Achieving Determinism in Adaptive AUTOSAR*.

3.4.1 Nondeterministic Emergency Brake Assistant

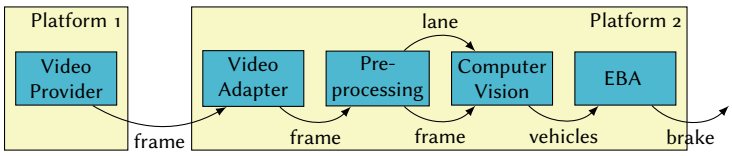


Figure 3.12: The emergency brake assistant (EBA) application implemented in the Adaptive Platform Demonstrator (APD).

AUTOSAR provides the Adaptive Platform Demonstrator (APD), which is an example implementation of the specification for AUTOSAR AP. It provides a set of demonstrator applications, where the most realistic and advanced application is the emergency brake assistant (EBA) shown in Figure 3.12. Unlike what may be expected of a safety-critical application, the brake assistant exhibits nondeterminism that could potentially have fatal consequences. While this demonstrator application is not designed for deployment in the real world, it illuminates the presence of uncontrollable and safety-undermining nondeterminism in AP.

The brake assistant consists of a pipeline of five SWCs, distributed across two platforms. Video Provider captures video frames and sends one approximately every 50 ms (via a proprietary protocol) to Video Adapter, which is running on the second platform. The communication along the remainder of the component chain occurs through AP service interfaces via the SOME/IP middleware. Event notifications are used to transfer data from one SWC to the next, and the corresponding event handler stores the data in a one-slot input buffer.

Each SWC configures a periodic callback so that the OS triggers the SWC logic every 50 ms. At each invocation, each component reads the current data item from its input buffer, performs some computation, and then communicates the result via an event.

For each frame that Preprocessing receives from Video Adapter, it computes a bounding box demarcating the current travel lane. Computer Vision receives from Preprocessing both the lane information as well as the original frame and uses them to detect vehicles in the lane and estimate their distance. It forwards the list of detected vehicles to the EBA component, which in turn decides whether an emergency brake maneuver is required.

The logic of each component processes the last data written to its one-slot input buffer. If there is no data, the SWCs silently stop computation and wait for the next periodic trigger to occur. This introduces nondeterminism as data could get overwritten before it is read by a downstream component, causing entire frames to be dropped. Moreover, since the Computer Vision component reads not one but two inputs, this can lead to misalignment between the video frames and the lane information. We instrumented the brake assistant code to detect and report frame loss and misalignment. Execution on an evaluation platform, consisting of two MinnowBoard Turbot[19] boards connected via an Ethernet switch, confirmed that the described errors indeed occur in a real-world setting. The boards are equipped with an Intel Atom E3845 Quad-Core processor and are officially supported by the APD.

19: MinnowBoard.org Foundation 2023, *MinnowBoard Turbot Dual Ethernet Family Technical Specs*.

A series of experiments was conducted to analyze the prevalence of the described errors. Each instance of the experiment executes the brake assistant and processes a total of 100,000 frames while counting dropped input values and misalignment between different inputs of the same component. Figure 3.13 plots the obtained results for a total of 20 experiment instances. Each bar in the figure shows the error prevalence for one instance of the experiment. The results are ordered by error rate for better visibility. This order does not correspond to the order in which results were obtained.

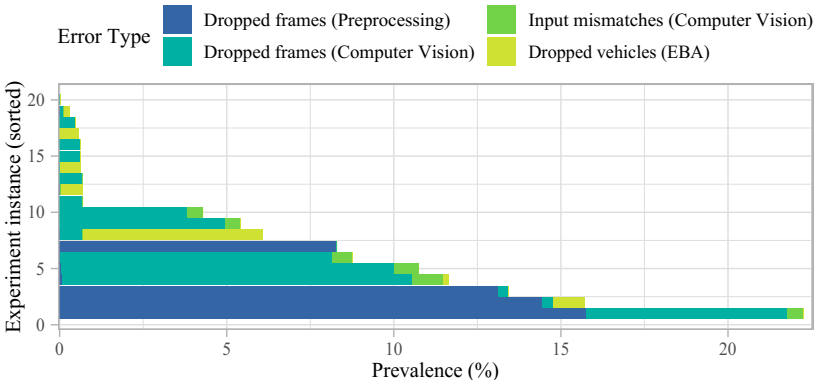


Figure 3.13: Prevalence of errors for 20 executions of the emergency brake assistant.

The error rate varies significantly between experiments. In the best case, the observed error rate is 0.018 % and in the worst case, the error rate is 22.25 %. The average error rate across all 20 experiments is 5.60 %.

Not only the overall error rate varies, but also the composition error types. In most experiments, frame dropping at Computer Vision is dominant, but for some experiments, dropped vehicles at EBA or dropped frames at Preprocessing dominate. This underlines the difficulty of assessing the performance and correctness of the brake assistant. The error rate is strongly influenced by the offset between the individual periodic callbacks of the SWCs, which depends on when the SWCs are started and is difficult to control.

In conclusion, the emergency brake assistant as implemented in the APD exposes errors due to nondeterminism that could potentially have fatal consequences. While a certain error rate might be acceptable for some applications, the presented experiments also highlight that the error rate itself is nondeterministic and influenced by parameters that are not part of the actual application design.

3.4.2 Deterministic Emergency Brake Assistant using Reactors

DEAR allows us to easily transform the brake assistant application into a deterministic reactor implementation. Since the original implementation separates computational logic from the communication mechanism, transformation requires only a few code changes. We encapsulate the logic of each SWC in a reactor that has one reaction to process incoming events. This reaction calls the original logic to process the data associated with the incoming event and produces an output event. The overall application is shown in Figure 3.14.

In order to support the transmission of tagged messages between SWCs, each reactor binds to the service interfaces of the SWC using the DEAR transactors. As described in Sections 3.3.2 and 3.3.3, a carefully chosen logical delay applied by the sending transactor ensures that messages are not sent at a physical time equal to or less than the message's timestamp. The receiving transactor further accounts for the physical delay of message transmission and ensures that incoming messages are only processed when it is safe to do so.

Since `ComputerVision` has two inputs, the reaction that calls the computer vision logic expects to receive two logically simultaneous messages. If only one input is present, this is considered an error. `VideoAdapter` has no well-defined input. It sporadically receives frames over the network sent by the camera. As the timing of the camera in this demonstrator application cannot

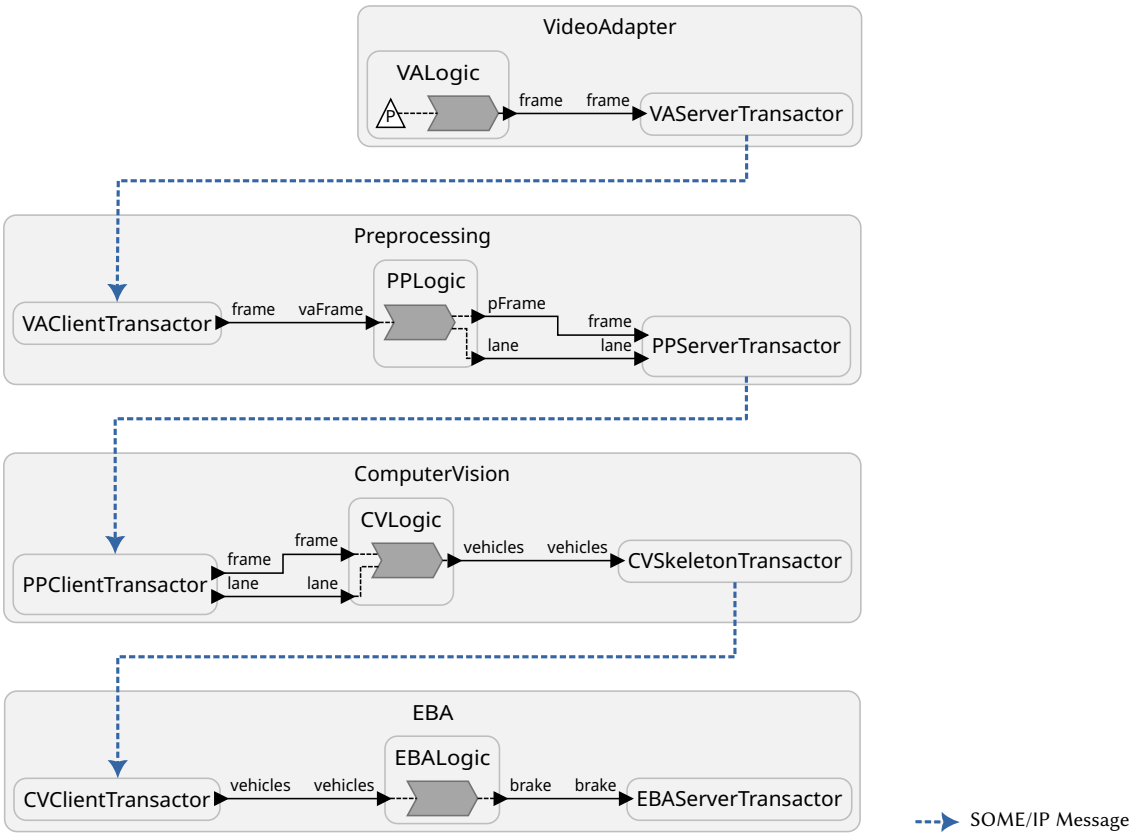


Figure 3.14: Deterministic implementation of the EBA application in the APD using reactors and the transactors provided by the DEAR framework.

be controlled, we implement `VideoAdapter`as a sensor that inserts frames into the reactor network with a tag equal to the physical time of message reception. Once the incoming frame is tagged, subsequent reactions are carried out in a deterministic order.

To achieve correct execution, it is important to carefully consider the physical delays imposed by the computations of each SWC and by the transport of messages between SWCs. Only if the delays associated with each SWC account for its WCET and if the specified maximum communication latency and the synchronization error are accurate, deterministic execution can be guaranteed. We use the values given in Table 3.1 for the deterministic EBA implementation. Since, in this application, all SWCs are deployed on the same platform, there is no clock synchronization error to account for. Note that these numbers are estimated upper bounds. More precise values can be obtained from WCET analysis.[20]

With this implementation, we achieve correct and deterministic execution on the MinnowBoard platform. Moreover, the timed semantics of reactors facilitates reasoning about the worst-case end-to-end latency between receiving a frame and producing an output brake signal.

20: Wilhelm et al. 2008, *The Worst-Case Execution-Time Problem-Overview of Methods and Survey of Tools*; Abella et al. 2015, *WCET Analysis Methods: Pitfalls and Challenges on their Trustworthiness*.

name	symbol	value
Video Adapter WCET	d_{VA}	5 ms
Preprocessing WCET	d_{P}	25 ms
Computer Vision WCET	d_{CV}	25 ms
EBA WCET	d_{EBA}	5 ms
communication latency	l	5 ms
clock synchronization error	c	0 ms

Table 3.1: Delays used in the deterministic EBA implementation.

These benefits come at the cost of an extra physical time delay, as each SWC has to account for worst-case computation and communication delays. This, however, is not a necessity. For certain applications, it is acceptable to deliberately introduce the possibility of sporadic errors by using delays lower than the actual WCET. Independent of how computation and communication delays are chosen, the reactor semantics guarantees determinism as long as the assumptions hold and translates any violation of one of the assumptions directly into observable errors.

In contrast to the original brake assistant implementation, which has nondeterministic error rates, the trade-off between end-to-end latency and error rate becomes apparent in the reactor implementation. Figure 3.15 shows results from an exemplary measurement on the `ComputerVision` component. The plot shows the correlation between the delay d_{CV} chosen for `ComputerVision` and the resulting error rate. This suggests, for example, that if the application can tolerate an error rate of about 2 %, then we could choose a much smaller delay of about $d_{\text{CV}} = 12$ ms. This trade-off between availability (or latency) and consistency (or error rate), is qualified in the CAL theorem (cf. Section 4.5.4).[21]

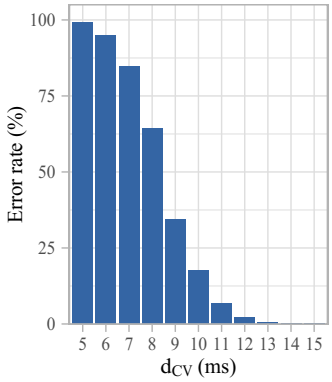


Figure 3.15: The trade-off between d_{CV} and the resulting error rate.

21: E. A. Lee, Bateni, et al. 2023, *Trading Off Consistency and Availability in Tiered Heterogeneous Distributed Systems*; E. A. Lee, Akella, et al. 2023, *Consistency Vs. Availability in Distributed Cyber-Physical Systems*.

3.5 Conclusion

This chapter introduced the reactor MoC, which combines principles known from other models like discrete events, actors, and synchronous languages to achieve deterministic concurrency. The chapter further demonstrated the usefulness of the reactor model for the development of CPS applications. Concretely, we introduced a C++ implementation of the reactor model as well as the DEAR framework, which facilitates the development of distributed deterministic applications on top of AUTOSAR AP. Based on the Adaptive Platform Demonstrator, we have shown that AP applications are inherently susceptible to nondeterminism and illustrated how the same application can be realized deterministically with reactors while maintaining standard compatibility.

The results presented in this chapter were published in Menard, Goens, Lohstroh, et al. 2020, *Achieving Determinism in Adaptive AUTOSAR*. Next to a C-based implementation for embedded devices,[22] the presented C++ runtime is one of the first published implementations of the reactor model. The case study based on the APD is the first published demonstration of the applicability of the reactor model to industrial use cases.

22: Lohstroh and E. A. Lee 2019b, *Work-in-Progress: Real-Time Reactors in C*.

Deterministic Coordination with Lingua Franca

4

The previous chapter introduced the DEAR framework, which comprises a C++ implementation of the reactor model. While this implementation facilitates the definition and execution of reactor programs, applications using DEAR are written in plain C++. However, C++ is an expressive general-purpose language, and programmers may easily introduce constructs that conflict with reactor semantics. Thus, to build correct programs, programmers need to be aware of the reactor semantics and disciplined to avoid accidental violations. Such violations could be (unprotected) interactions with concurrent threads, accessing global shared state, directly accessing the state of other reactors, or accessing ports and actions that have not been declared as a dependency of a reaction.

If programmers do not fully understand the semantics of the underlying MoC, they are tempted to take shortcuts and break out of the model to achieve certain goals, which can undermine the benefits of using the model. This was shown, for instance, in a comprehensive study of programs using Scala actors.[1]

Relying on programmers' discipline is a common practice in the industry. The automotive industry, for instance, has introduced standardized guidelines for C and C++ programming to improve the reliability of automotive software.[2] However, a better approach is to utilize languages that provide stricter guarantees and where certain properties can be statically checked by the compiler. Rust, for instance, eliminates the memory safety issues that are ubiquitous in C/C++ by utilizing *ownership types*[3] to allow the compiler to reason about a program's memory accesses and to guarantee correctness.[4] For this reason, Rust is becoming increasingly popular in the fields of embedded systems, operating systems, and safety-critical systems, which have been mostly dominated by C/C++ in the past.

In order to provide a language that allows the specification of deterministic concurrent software, we created the coordination language Lingua Franca (LF). Lingua Franca builds on the principles of the reactor model and implements a correct-by-construction approach. The LF compiler ensures the correctness of given programs and, to the extent possible, that there are no accidental or deliberate violations of the reactor semantics. Thus, LF can provide better guarantees on the correctness of a program than a plain C++ implementation using the DEAR framework or a similar implementation of reactors. In addition, more convenience features and utilities can be provided on the level of a domain-specific language.

This chapter introduces Lingua Franca and gives a broad overview of the language, the tooling, and the research conducted. Lingua Franca is a largely collaborative effort with many contributors. Most ideas and concepts cannot be attributed to a single person. The author of this thesis contributed the C++ target to Lingua Franca and made significant contributions to the development of Lingua Franca as a whole. Therefore, this chapter aims to give a complete overview of Lingua Franca with a focus on the specifics of the C++ target.

Multiple publications exist that discuss various aspects of the Lingua Franca language. Some of the examples and discussions presented in this chapter were previously published. This includes:

1: Tasharofi, Dinges, and Johnson 2013, *Why Do Scala Developers Mix the Actor Model with other Concurrency Models?*

2: AUTOSAR 2019, *Guidelines for the Use of the C++14 Language in Critical and Safety-related Systems*; The MISRA Consortium et al. 2023, *MISRA C:2023: Guidelines for the Use of the C Language in Critical Systems*.

3: Boyapati, R. Lee, and Rinard 2002, *Ownership Types for Safe Programming: Preventing Data Races and Deadlocks*.

4: Klabnik and Nichols 2022, *The Rust Programming Language*.

Figure 4.1: The Lingua Franca logo.

- Lohstroh, Menard, Schulz-Rosengarten, et al. 2020, *A Language for Deterministic Coordination Across Multiple Timelines*
- Lohstroh, Menard, Bateni, et al. 2021, *Toward a Lingua Franca for Deterministic Concurrent Systems*
- Menard, Lohstroh, et al. 2023, *High-Performance Deterministic Concurrency Using Lingua Franca*
- Lohstroh, Bateni, et al. 2023, *Deterministic Coordination Across Multiple Timelines*.

4.1 Polyglot Coordination

Lingua Franca is a polyglot coordination language that is not intended to be a general-purpose programming language. Therefore, the syntax of LF focuses on describing the structure of reactor programs and does not provide mechanisms for describing the business logic of reactors. Implementation of the logic is delegated to one of the available target languages. At the moment of this writing, LF supports C, C++, Rust, Python and TypeScript as target languages. Each LF program specifies its target language at the beginning and embeds target code blocks that implement the business logic. Consider, for instance, Listing 4.1, which shows a "Hello, World!" program in Lingua Franca for all the available target languages.

Listing 4.1: "Hello, World!" programs written in LF using all the target languages that are currently supported.

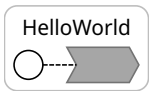
```
target C
main reactor{
  reaction(startup) {=
    printf("Hello, World\n");
  =}
}
```

```
target Cpp
main reactor{
  reaction(startup) {=
    std::cout <<"Hello, World\n";
  =}
}
```

```
target Rust
main reactor{
  reaction(startup) {=
    println!("Hello, World");
  =}
}
```

```
target Python
main reactor{
  reaction(startup) {=
    print("Hello, World")
  =}
}
```

```
target TypeScript
main reactor{
  reaction(startup) {=
    console.log("Hello, World!")
  =}
}
```



This approach is different from many related languages. In the synchronous programming languages, for instance, the entire business logic needs to be implemented using the synchronous-reactive paradigm. LF, instead, clearly separates the coordination of concurrent components from their business logic. By incorporating target code blocks, LF can easily integrate with existing third-party libraries and legacy code bases.

The target code blocks are treated as black boxes. This approach is enabled by the clearly defined interfaces of reactions in the reactor model. Reactions need to declare their dependencies and anti-dependencies, and we can organize all reactions in an APG that captures all scheduling constraints (cf. Section 3.1.3). This graph is valid regardless of the concrete code that gets executed when triggering a reaction.

Lingua Franca programs are not directly compiled into executable code. Instead, the LF compiler generates code in the target language. A target toolchain is responsible for compiling the executable and linking it with a reactor runtime that coordinates the execution of the LF program. This process is depicted in Figure 4.2. This two-step approach allows for a separating concerns. The LF compiler is responsible for checking the correctness of the program regarding the semantics of the reactor model and for generating target code that correctly implements the reactor program. The target

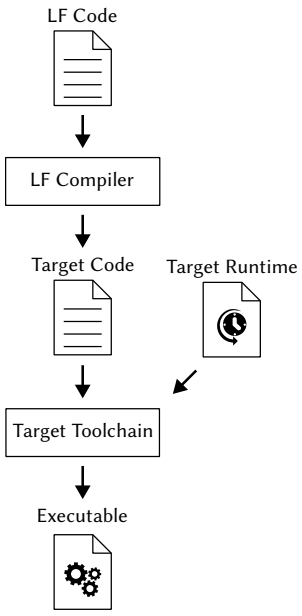


Figure 4.2: The LF compilation flow.

toolchain, however, is responsible for type-checking and validating target code blocks.

4.2 Syntax

The syntax of Lingua Franca is closely based on the reactor model. Each of the reactor elements is represented as a keyword or operator in LF. However, LF also provides various extensions over the reactor model, which can be seen as "syntactic sugar." Listing 4.2 lists the core syntax rules of Lingua Franca using the ANTLR 4 format.[5] In the following, we discuss these syntax rules in more detail to provide an overview of the Lingua Franca coordination language. The underlying concepts and code examples are discussed in the subsequent sections.

5: Parr 2013, *The Definitive ANTLR 4 Reference*.

Each Lingua Franca program consists of a target declaration, an optional list of imports, an optional set of preambles, and at least one reactor declaration (line 3). The target declaration starts with the `target` keyword and specifies the target language of the program (line 5). Optionally, the target declaration may specify additional target properties. Such properties can influence the behavior of the program or its compilation.

Following the target declaration, the program may specify one or multiple imports using the `import` and `from` keywords. Each import statement specifies one or multiple reactors to import, as well as the path to an LF file to import from (line 8).

The program may provide one or more preambles in the file-global scope (line 10). Preambles allow the programmer to specify additional imports of third-party libraries in the target language, or to define global variables and functions. Preambles are declared using the `preamble` keyword and a target language code block. In LF, such target code blocks are enclosed in {= and =} (line 50). Depending on the target language, the preamble may be additionally qualified using the `private` or `public` keywords. This is relevant for target languages like C++, where declarations are placed publicly in a header file and definitions are provided in an additional source file.

The main section of each LF program consists of at least one reactor declaration (lines 12–15). Reactors are declared using the `reactor` keyword. Each reactor may declare a set of type parameters and parameters. A reactor contains any number of reactor-level preambles, state variables, methods, ports, timers, actions, reactor instantiations, connections, and reactions. Reactor-level preambles are similar to the file-level preambles, but their scope is limited to the contents of the reactor.

Reactors may be qualified using the `main` or `federated` keywords. The main reactor represents the top-level reactor of an LF program. The `federated` keyword also marks the top-level reactor, but in contrast to `main`, `federated` implies that the program is compiled into multiple binaries that may be executed in a distributed system. The LF compiler generates one executable for each reactor instantiated within the federated reactor. The name of a reactor qualified as `main` or `federated` may be omitted. If it is given, it is required to be equal to the file name of the LF program.

Parameters and state variables are defined by their name, an optional type, and an optional initializer expression (lines 20 and 21). State variables are declared using the `state` keyword. Parameters do not require a dedicated keyword and are declared syntactically as arguments to the reactor. Parameters are an extension that LF provides over the basic reactor model. They can

Listing 4.2: The core grammar of Lingua Franca given in ANTLR 4 format.

```
grammar LinguaFranca;

program: target lf_import* preamble* reactor+ EOF;

target: 'target' ID ('{' property* '}')?;
property: key=ID ':' value=(literal | time);

lf_import: 'import' reactors+=ID (',' reactors+=ID)* 'from' uri=STRING;

preamble: (qualifier=('private' | 'public'))? 'preamble' code;

reactor:
    ('main' | 'federated')? 'reactor' (name=ID)? type_parameters? parameters?
    '{' ( preamble | state | method | port | timer | action | instantiation | connection | reaction)* '}'
;

type_parameters: '<' ID (',' ID)* '>'
parameters: '(' parameter (',' parameter')')*

parameter: name=ID (':' type)? '=' (init=expression)?;
state: 'state' name=ID  (':' type)? '=' (init=expression)?;

method: 'const'? 'method' name=ID '(' (method_argument (',' method_argument)*)? ')' (':' return=type)? code;
method_argument: name=ID (':' type)?;

port: ('input' | 'output') name=ID (':' type)?;
timer: 'timer' name=ID ('(' offset=expression (',' period=expression)? ')')?;
action: ('logical' | 'physical') 'action' name=ID (':' type)?;

instantiation: name=ID '=' 'new' reactorClass=ID '(' (assignment (',' assignment)*)? ')';
assignment: parameter_name=ID '=' expression;

connection: reference ('->' | '~>') reference ('after' delay=expression)?;
reference: variable=ID | container=ID '.' variable=ID;

reaction:
    'reaction' name=ID? ('(' (triggers+=trigger_reference (',' triggers+=trigger_reference)*)? ')')
    (sources+=reference (',' sources+=reference)*)? ('->' effects+=reference (',' effects+=reference)*)?
    code deadline?
;

trigger_reference: 'startup' | 'shutdown' | reference;

deadline: 'deadline' '(' delay=expression ')' code;

type: 'time' | ID | code;

expression: code | literal | parameter_ref | time;

code: '{=' .*? '=}';
literal: BOOLEAN | INT | NEGINT | FLOAT | STRING;
parameter_ref: ID;
time: value=INT unit=ID?;

WHITESPACE : [ \t\r\n]+ -> skip;
INT: ('0'..'9')+;
NEGINT: '-' ('0'..'9')+;
BOOLEAN: 'true' | 'True' | 'false' | 'False';
ID: '^'? ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;
STRING:
    '"' ( '\\' ('b' | 't' | 'n' | 'f' | 'r' | 'u' | '"' | '\'' | '\\') | ~('\\' | '"') )* '"' |
    '\'' ( '\\' ('b' | 't' | 'n' | 'f' | 'r' | 'u' | '"' | '\'' | '\\') | ~('\\' | '\'') )* '\'';
FLOAT:  (INT | NEGINT)? '.'  INT (('e' | 'E') ('+' | '-')? INT)?;
ANY_OTHER: .;
```

be seen as constant state variables that are initialized once when a reactor is instantiated.

When targeting strongly typed languages, the LF compiler enforces that a type is provided. A type in LF is either given using a simple identifier, using target code, or using the built-in type `time` (line 46). `time` denotes a time span and is the only type that is directly interpreted and checked by the LF compiler.

In addition to regular parameters, a reactor may also declare type parameters enclosed in < and > (line 17). Type parameters allow the definition of generic reactors that are similar to generic classes in C++ or Java. A concrete type needs to be specified when instantiating a generic reactor.

Expressions in LF are either given as target code blocks, literal values, references to parameters, or time values (line 48). Literals are Boolean values, signed and unsigned integers, floating-point numbers, and strings (line 51). This rule covers most literals found in LF's target languages. LF provides more syntax rules that match additional literals and common expressions found in various target languages, e.g., list expressions in Python, but such rules are omitted here for brevity. Any expression that does not match the LF syntax rules can simply be given as a target code expression enclosed in {= and =}. Expressions may also refer to a reactor parameter or denote a time value. A time value in LF is a pair of an integer literal and a unit (line 53). If the value is zero, then the unit may optionally be omitted. All time values are of type `time`.

Methods are another extension of LF over the basic reactor model. Similar to private methods in object-oriented programming (OOP), methods in LF can access the local reactor state and provide a mechanism for reusing functionality within the reactor scope. Methods may be called from reaction bodies or from other methods. The `method` declaration specifies an identifier, an optional return type, and a set of typed and named arguments (lines 23 and 24). The method body is given as a target code block. The additional `const` qualifier can be used to mark the method as read-only[6] for target languages that support this functionality.

6: Here, read-only means that the method can read state variables but not write them.

Ports, timers, and actions can be declared natively in LF. Ports are declared using the `input` or `output` keywords followed by an identifier and an optional type annotation (line 26). In strongly typed languages, the type is required.

Timers are declared using the `timer` keyword and an identifier (line 27). Timers are optionally configured with an offset and a period. The offset specifies the logical delay until the timer's first triggering after program startup, and the period specifies the logical delay between subsequent triggerings. If the period is omitted or set to zero, then the timer triggers only once. If the offset is omitted or set to zero, then the first triggering is logically simultaneous with the startup event.

Actions are declared using the `action` keyword, an identifier, an optional type, and either the `logical` or the `physical` qualifier (line 28). Some targets in LF support additional configurations for actions. For instance, it is possible to specify a minimum delay or an inter-arrival time. However, the syntax rules for such configurations are omitted in this exposition for brevity.

Reactors are instantiated using the assignment operator = and the `new` keyword (line 30). Each instantiation needs to specify an instance name and refer to a concrete reactor declaration to instantiate. If the reactor that gets instantiated has parameters, then the instantiation may specify one or multiple parameter assignments (line 31).

Connections are created using the -> or ~> operators (line 33). Each connection specifies a reference to a port on the left-hand side and on the right-hand side of the operator. The port reference either refers to a port of the local reactor or to a nested port within a reactor instance using a dot separator (line 34). While the -> operator creates a regular connection such that input and output are logically simultaneous (as defined in the reactor model), the ~> operator creates a physical connection. A physical connection reassigns the timestamp based on the current reading of physical time when receiving a message.[7] In addition, the **after** keyword may be used to specify a logical delay that the connection imposes. After delays and physical connections are explained in more detail in Sections 4.4.5 and 4.4.7.

7: This makes physical connections analogous to physical actions.

Finally, reactions are declared using the **reaction** keyword (lines 36–40). Reactions may optionally be named. Since reactions are triggered by events and executed by the runtime, they may not be called directly. The reaction name is a purely cosmetic annotation. Each reaction specifies a set of triggers, sources and effects. Triggers can be any input port, timer or action within the scope of the reactor, output ports of contained reactors, or the keywords ***startup*** and ***shutdown***, which represent the startup and shutdown triggers of the reactor model. Sources are ports or actions that the reaction may read from, but that do not trigger the execution of the reaction. Effects are any output ports or actions within the scope of the reactor or the input ports of contained reactors. A reaction may write to its effect, i.e., it may set any ports or schedule any actions that are declared to be an effect of the reaction.

The reaction body is given as a target code block. The **deadline** keyword may optionally be used to annotate the reaction with a deadline. This specifies an upper bound in physical time fo when the reaction executes relative to its tag. If the deadline is violated, a deadline handler is executed instead of the reaction body. This handler is also given as a target code block. See Section 4.4.3 for more details on deadlines in LF.

4.3 Code Examples

While the previous section introduced the syntax rules of LF, any discussion of programming languages requires code examples to better illustrate the fundamental concepts. This section reconsiders the bank account example programs that were introduced in Section 3.2 and discusses their implementation in LF code.

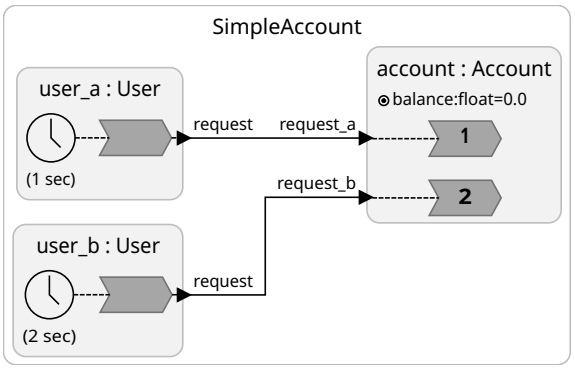
4.3.1 Simple Bank Account Example

Consider the program given in Listing 4.3. It provides an LF implementation of the reactor example initially given in Figure 3.5 on Page 48. The program defines three reactors: `User`, `Account`, and the main reactor that assembles the program.

`User` is defined on lines 3–10. It is parameterized by an `offset` of type **`time`** and a `value` of type **`float`** It consists of a timer `t` that triggers once at `offset` and an output port `request` of type **`float`**. The reaction defined on line 7 is triggered by the timer `t` and may have an effect on the `request` output port. The reaction body is given in C++ code; it simply sends a request with a value as given by the `value` parameter. Thus, any instance of the `User` reactor sends a single request for the given `value` at the specified logical offset.

Listing 4.3: Lingua Franca implementation of the account example given in Figure 3.5.

```
1  target Cpp
2
3  reactor User(offset:time = 0, value:float = 0.0) {
4    timer t(offset)
5    output request: float
6
7    reaction(t) -> request {=
8      request.set(value);
9    =}
10 }
11
12 reactor Account {
13   state balance: float = 0.0
14   input request_a: float
15   input request_b: float
16
17   reaction(request_a) {=
18     apply(*request_a.get());
19   =}
20
21   reaction(request_b) {=
22     apply(*request_b.get());
23   =}
24
25   method apply(value:float) {=
26     std::cout <<"Request of" << value <<" was ";
27     if (balance + value >= 0) {
28       balance += value;
29       std::cout <<"accepted\n";
30     } else {
31       std::cout <<"denied\n";
32     }
33   =}
34 }
35
36 main reactor {
37   offset_a:time = 1s,
38   offset_b:time = 2s
39 ) {
40   account = new Account()
41   user_a = new User(offset=offset_a, value=20)
42   user_b = new User(offset=offset_b, value=-10)
43
44   user_a.request -> account.request_a
45   user_b.request -> account.request_b
46 }
```



`Account` is defined on lines 12–34. It contains a state variable called `balance` of type **`float`** that is initialized with zero. This variable represents the current balance of the account. `Account` further contains two input ports of type **`float`** called `request_a` and `request_b`. These ports allow the account to receive requests from two separate users. The two reactions defined on lines 17 and 21 react to incoming requests. Both reactions obtain the requested value and then call the `apply` method, which implements the business logic of the account.[8] In the C++ target, values are obtained by calling `get()` on a port reference, which returns a smart pointer[9] to the current value of the port. To obtain the actual value, this pointer is dereferenced using the `*` operator (cf. Section 4.7.1). The `apply` method simply tests if the request can be applied without getting a negative balance. If it can be applied, it updates the balance and prints "accepted". Otherwise, it prints "denied".

The main reactor assembles the program. It creates a single instance of `Account` (line 40) and two instances of `User` (lines 41, 42), and connects the outputs of the user instances to the inputs of the account instance (lines 44, 45) using the connection operator `->`. `user_a` is parameterized with an offset of 1 s and a value of 20 and `user_b` is parameterized with an offset of 2 s and a value of -10.

When executed, the program prints the following output:

```
1 Request of 20.0 was accepted
2 Request of -10.0 was accepted
```

1 s after startup, `user_a` sends a deposit request with a value of 20, which the account accepts. 2 s after startup, `user_b` sends a withdrawal request with a value of -10, which the account also accepts.

8: Here, we use two distinct ports and reactions to keep the discussion simple. Such an approach, however, is not very practical as we cannot easily change the number of users and the reactions duplicate logic. Section 5.1 introduces a syntax extension that allows for a more flexible and compact definition of `Account`.

9: Edelson 1992, *Smart Pointers: They're Smart, But They're Not Pointers*; Dmitrović 2023, *Smart Pointers*.

In the C++ target, main reactor parameters are automatically synthesized into command-line arguments that allow overwriting the parameter values at startup. Using the parameters `offset_a` and `offset_b`, we can change the order in which the users send the deposit and withdrawal requests. For instance, running the command

```
./bin/SimpleAccount --offset_a 2s --offset_b 1s
```

reverses the order of the two user requests. The program prints:

```
Request of -10.0 was denied
Request of 20.0 was accepted
```

We can also use the same offset for both users. For instance, running the command

```
./bin/SimpleAccount --offset_a 0 --offset_b 0
```

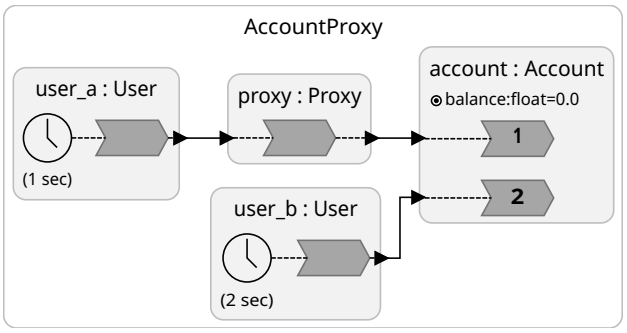
prints:

```
Request of 20.0 was accepted
Request of -10.0 was accepted
```

In this case, the two requests are logically simultaneous. However, the order in which the two requests are processed is uniquely defined by the APG. Since the reaction to `request_a` is defined before the reaction to `request_b`, the reaction to `request_a` takes precedence over the reaction to `request_b`. Hence, the deposit request is processed before the withdrawal request, even if both are logically simultaneous.

Listing 4.4: Lingua Franca implementation of the account example with a proxy reactor given in Figure 3.6a.

```
target Cpp

import Account, User from "SimpleAccount.lf"

reactor Proxy {
  input in: float
  output out: float

  reaction(in) -> out {=
    out.set(*in.get() - 5);
  =}
}

main reactor {
  account = new Account()
  user_a = new User(offset=0s, value=20)
  user_b = new User(offset=0s, value=-10)
  proxy = new Proxy()

  user_a.request -> proxy.in
  proxy.out -> account.request_a
  user_b.request -> account.request_b
}
```



4.3.2 Bank Account Examples with Proxy Reactors

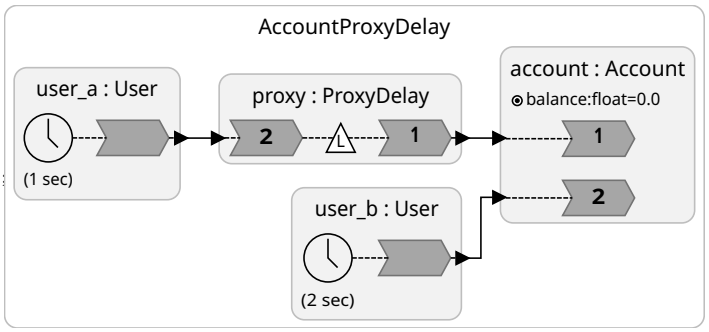
As discussed in Section 3.2, inserting a simple proxy reactor does not change the order of events. The code given in Listing 4.4 reproduces the reactor program given in Figure 3.6a on Page 49. It introduces a proxy reactor between `user_a` and `account`. While the proxy reactor may modify the request (in this case, it subtracts 5 from the request value before forwarding the request), it does not influence the logical ordering of events. Input and output of the proxy reactor are logically simultaneous. When executed, the program produces the following output:

```
Request of 15 was accepted
Request of -10 was accepted
```

As also discussed in Section 3.2, we can deliberately introduce a logical delay using a logical action. The code given in Listing 4.5 reproduces the reactor program given in Figure 3.6b. The program defines a reactor called `ProxyDelay` that has a parameter `delay` of type `time` and a type parameter `T`. `ProxyDelay` contains an input port, an output port, and a logical action, all of which are of type `T`. This pattern makes the reactor generic and allows for the forwarding of arbitrary message types.

Listing 4.5: Lingua Franca implementation of the account example with a proxy reactor imposing a delay as given in Figure 3.6b.

```
1  target Cpp
2
3  import Account, User from "SimpleAccount.lf"
4
5  reactor ProxyDelay<T>(delay:time = 0) {
6    input in: T
7    output out: T
8    logical action a: T
9
10   reaction(a) -> out {=
11     out.set(std::move(a.get()));
12   =}
13
14   reaction(in) -> a {=
15     a.schedule(std::move(in.get()), delay);
16   =}
17 }
18
19 main reactor(delay:time=2s) {
20   account = new Account()
21   user_a = new User(offset=0s, value=20)
22   user_b = new User(offset=0s, value=-10)
23   proxy = new ProxyDelay<float>(delay=delay)
24
25   user_a.request -> proxy.in
26   proxy.out -> account.request_a
27   user_b.request -> account.request_b
28 }
```



The reaction on line 10 reacts to the logical action and simply forwards the value stored on the action to the output port. The C++ code in the reaction body uses `std::move` to enforce move semantics and avoid copying the underlying smart pointers. The reaction on line 14 reacts to the input port and schedules the logical action using the specified delay. Thus, any message received on the input port will result in a later message on the output port, such that the logical delay between input and output is precisely `delay`.

Line 23 instantiates the `ProxyDelay` reactor using a default delay of 2 s and the type `float`. Therefore, the program delays any requests sent by `user_a` by 2 s. Consequently, the program denies the withdrawal request by `user_b` and produces the following output:

```
Request of -10 was denied
Request of 20 was accepted
```

We can change the program's behavior by modifying the `delay` parameter. Running `./bin/AccountProxyDelay --delay 500ms` will instead print:

```
Request of 20 was accepted
Request of -10 was accepted
```

4.4 Coordinating Logical and Physical Time

All the examples discussed in the previous subsection are deterministic. All events are ordered logically regardless of the physical time required for executing reactions. While all events are timed, the physical timing for handling events is approximate. This section discusses how LF programs relate the logical timeline of events with the physical wall clock time.

4.4.1 Timing Diagrams

To illustrate the execution of LF programs and the relations between logical and physical time, we use *timing diagrams* like the one shown in Figure 4.3.[10] The x-axis of a timing diagram denotes physical time, and the y-axis denotes logical time. For simplicity, we will assume that each program starts at physical time $T_0 = 0$ with tag $g_0 = (0, 0)$.

10: This form of representing the execution of LF programs is inspired by a similar visualization proposed by Erling Jellum.

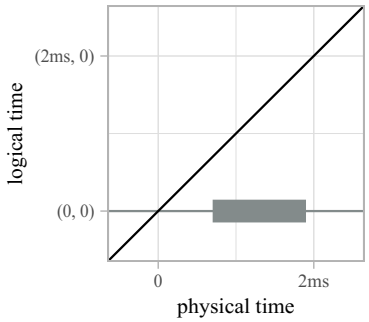


Figure 4.3: Timing diagram illustrating a possible execution of the "Hello, World!" LF program given in Listing 4.1.

A vertical gray line indicates the presence of an event at the corresponding tag, and a vertical bar indicates the execution of a reaction. The position of the bar illustrates when the reaction starts executing (in physical time), and the length of the bar illustrates the time required for processing the reaction.

The diagonal black line denotes the function $\mathcal{G} : \mathbb{T} \to \mathbb{G}$ with $\mathcal{G}(t) = (t, 0)$. As discussed in Section 3.1.4, the runtime scheduler only processes an event with tag g, once the current reading of physical time T is greater than the timestamp of g ($T > \mathcal{T}(g)$). Or, in other words, if the tag $\mathcal{G}(T)$ corresponding to the current reading of physical time is greater than the tag g ($\mathcal{G}(T) > g$). Therefore, all bars indicating the execution of a reaction need to start on the right side of the diagonal line. The crossing of a vertical line that represents an event with the black diagonal line marks the earliest physical time at which the event may be processed.

Consider again the "Hello, World!" programs given in Listing 4.1. This program has a single event (***startup***) and executes a single reaction. Figure 4.3 illustrates one possible execution trace for this program. The gray horizontal line denotes the startup event, and the gray bar corresponds to the execution of the reaction. The gap between the bar and the diagonal line represents the scheduling delay. This is the time it takes to wake up the scheduler, mark the reaction for, execution, and start processing the reaction. Depending on the concrete hardware and OS, this gap might be narrower or wider, and it might be more or less affected by jitter.

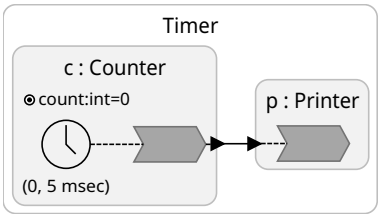
4.4.2 Physical Time Barrier and Fast Execution

The timing diagram can be read like a Gantt chart. The program proceeds to the right in physical time as observed by the local physical clock. For each physical time instance, the diagram shows which reactions execute. However, the timing diagram also shows the progression of logical time on the vertical axis. When the reactor runtime decides to process the next event, the execution jumps up to the next tag. Thereby, the black diagonal line denotes the physical time barrier implemented by the runtime, which prevents logical time from progressing faster than physical time.

Consider the program given in Listing 4.6. The program consists of two reactors. The `Counter` reactor keeps a state variable `count`, which it increments at every tick of the timer `t`, i.e., every 5 ms. It also sends the current value of `count` via its output port. This value is received by the `Printer` reactor,

Listing 4.6: A Lingua Franca program that increments a counter every 5 ms and prints the current value.

```
target Cpp

reactor Counter {
  state count: int = 0
  timer t(0, 5 ms)
  output out: int

  reaction(t) -> out {=
    out.set(count++);
  =}
}


reactor Printer {
  input in: int

  reaction(in) {=
    std::cout <<"Received:"
              << *in.get() <<'\n';
  =}
}

main reactor{
  c = new Counter()
  p = new Printer()
  c.out -> p.in
}
```



which simply prints the value. When executed, the program produces the following output:

```
Received: 0
Received: 1
Received: 2
Received: 3
Received: 4
...
```

A new line appears roughly every 5 ms of wall clock time.

Figure 4.4a shows a timing diagram that visualizes one possible execution trace of this program for the first 5 events. With each event produced by the timer, the execution progresses to the right and to the top within the diagram. It progresses to the right as physical time advances and progresses to the top as the scheduler advances logical time to the next tag. This diagram illustrates how the physical time barrier governs program execution. After executing both reactions, the scheduler waits until physical time catches up with logical time before processing the next event.

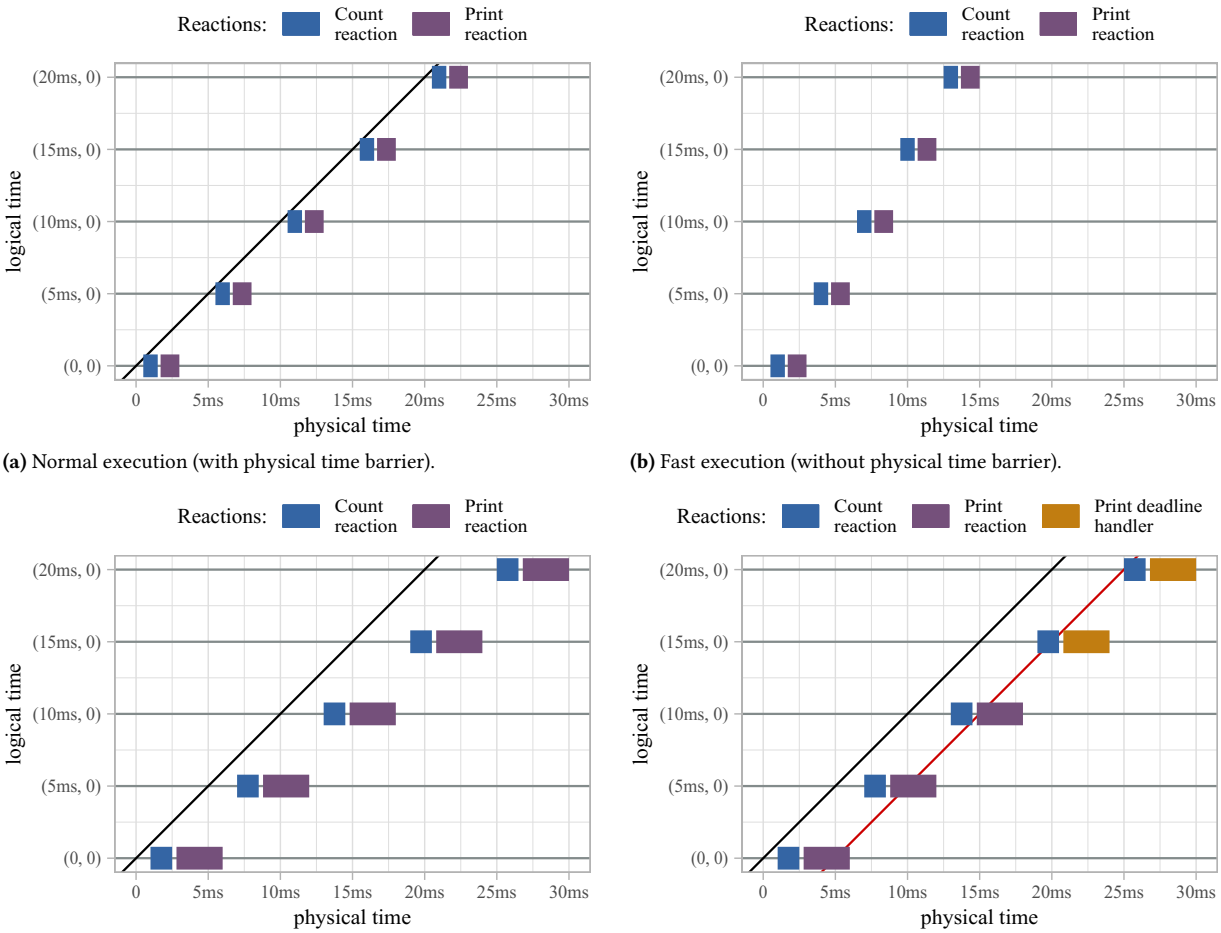
In LF, the physical time barrier may optionally be switched off. This is referred to as the *fast mode*. The fast mode can be activated by setting the target property `fast: true` in the LF program. In the C++ target, the fast mode can also be activated at runtime using the `--fast` command line argument.

Figure 4.4b shows the execution of the program from Listing 4.6 in fast mode. The advancement of logical time is not governed by the physical time barrier. Instead, the scheduler processes the next event as soon as all reactions at the current tag are completed.

4.4.3 Lag and Deadlines

While the physical time barrier presents a lower bound for when a reaction to an event at a given tag is executed, there is no upper bound. Lingua Franca implements a best-effort approach that aims at executing reactions at a physical time that is close to the timestamp of their tag, but it does not guarantee that the delay between timestamp and execution is bounded.

If logical time advances slower than physical time, we say that it *lags* behind physical time. Lag may occur, for instance, when reactions take longer to execute than indicated by the logical delay between events. Figure 4.4c shows the timing diagram for an alternative execution trace of the program in Listing 4.6. Compared to the timing diagram in Figure 4.4a, the reactions require more time to execute. There is an increasing lag since the reaction execution time is larger than the logical delay of 5 ms between events.



(a) Normal execution (with physical time barrier).

(b) Fast execution (without physical time barrier).

(c) An execution that lags behind logical time.

(d) A lagging execution with an annotated deadline.

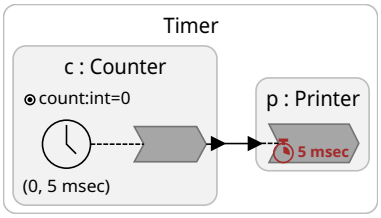
Figure 4.4: Timing diagrams visualizing possible execution traces for the program given in Listing 4.6.

While LF does not guarantee the absence of lag, it does provide a mechanism for detecting lag. Reactions may optionally be annotated with a deadline and a deadline handler. Consider the modified implementation of the `Printer` reactor given in Listing 4.7. It uses the **`deadline`** keyword to annotate a deadline of 5 ms. If the deadline is violated, the deadline handler will be executed instead. In this example, the handler simply prints `"Deadline violated"`!

A deadline in LF does not impose an upper bound on when a reaction finishes its computation, but instead imposes an upper bound on when a reaction starts its computation. At the current tag g, a deadline D is considered violated if the corresponding reaction starts executing at a physical time T with $T > \mathscr{T}(g) + D$. If the deadline is violated, then the deadline handler is executed instead of the reaction body. This mechanism makes lag detectable and also handleable by the business logic. A deadline violation is simply treated as a fault that can be handled as indicated by the user.

Figure 4.4d shows the timing diagram for the modified program with an annotated deadline. The diagonal red line represents the deadline. More precisely, it displays the function $\mathfrak{D}_D : \mathbb{T} \to \mathbb{G}$ defined by $\mathfrak{D}_D(T) = (T - D, 0)$, which maps a given reading of physical time T to the latest tag $g = \mathfrak{D}_D(T)$, that can be executed at this time without violating the deadline condition $T \leq \mathscr{T}(g) + D$.

```
reactor Printer {
  input in: int

  reaction(in) {=
    std::cout <<"Received:"
              << *in.get() <<'\n';
  =} deadline(5 ms) {=
    std::cout <<"Deadline"
              << "violated\n";
  =}
}
```



Listing 4.7: Modified `Printer` reactor with an annotated deadline and a deadline handler.

In Figure 4.4d, there is an increasing lag, similar to the diagram in Figure 4.4c. For the first 3 iterations, the reaction of `Printer`is executed within the deadline. However, stating from the 4th iteration, the deadline is violated, and the deadline handler is executed instead of the reaction body. The program prints:

```
Received: 0
Received: 1
Received: 2
Deadline violated!
Deadline violated!
...
```

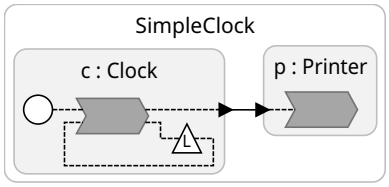
4.4.4 Logical Actions

While ports provide a mechanism for triggering simultaneous events at the current tag in connected reactors, actions provide a mechanism for scheduling events at a later tag, but only within the scope of one reactor. As discussed in Section 3.1.2, logical actions are scheduled with a tag relative to the current tag. In LF, reactions may declare logical actions as effects, and the reaction body may schedule the declared actions using an optional delay. Logical actions can be used to implement logical delays, as was already shown in Listing 4.5. If the delay is omitted or set to zero, then the new event is scheduled at the next microstep.

Logical actions may also be scheduled repeatedly. Consider Listing 4.8, which implements a simple clock. The `Clock`reactor defines an output port `tick` and a logical action a. The reaction on line 7 is triggered by the ***startup*** event, or a, and may have an effect on the action a and the output port `tick`. The reaction body schedules the next triggering of a with a delay equal to the given period, and it sets the output port `tick`at the current tag. This pattern effectively implements a timer with zero offset. Hence, timers in LF can be considered syntactic sugar.

Listing 4.8: A simple clock implemented in LF that sends a tick signal at regular intervals.

```
1  target Cpp
2
3  reactor Clock(period: time = 0) {
4    output tick: void
5    logical action a
6
7    reaction(startup, a) -> a, tick {=
8      a.schedule(period);
9      tick.set();
10   =}
11 }
12
13 reactor Printer {
14   input tick: void
15   reaction(tick) {=
16     std::cout <<"Tick at "
17       << get_elapsed_logical_time()
18       << '\n';
19   =}
20 }
21
22 main reactor {
23   period: time = 100 ms
24 ) {
25   c = new Clock(period=period)
26   p = new Printer()
27   c.tick -> p.tick
28 }
```



When executed, the program prints:

```
Tick at 0 nsecs
Tick at 100000000 nsecs
Tick at 200000000 nsecs
Tick at 300000000 nsecs
Tick at 400000000 nsecs
...
```

Figure 4.5 shows the corresponding timing diagram, which is similar to the diagram in Figure 4.4a.

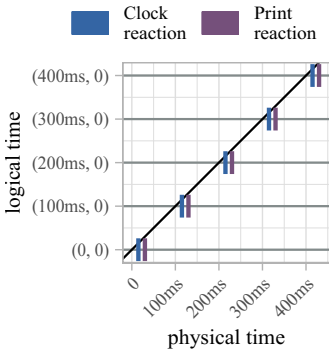


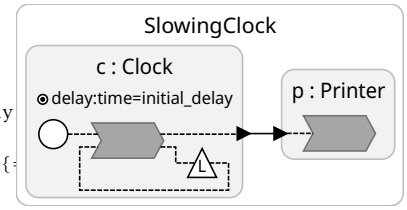
Figure 4.5: Timing diagram for the simple clock program in Listing 4.8.

Timers are more convenient to use than logical actions, when reactions should be triggered at regular intervals. However, logical actions provide more flexibility, as the target code in the reaction body can freely choose if it schedules the action and which delay to use. We can use this, for instance, to implement a clock with a changing period.

Consider, for example, the program in Listing 4.9 that implements a slowing clock. The `Clock`reactor in this example implements a variable delay using the state variable defined on line 23. Similar to the simple clock example, the body of the reaction declared on line 25 schedules the action a. However, it uses a variable `delay`, which is incremented by `step` for each reaction invocation.

Listing 4.9: A slowing clock implemented in LF that sends a tick signal at increasing intervals.

```
1  target Cpp
2
3  import Printer from "SimpleClock.lf"
4
5  main reactor {
6    step: time = 100 ms,
7    initial_delay:time = 100 ms
8  ) {
9    c = new Clock(
10     step=step,
11     initial_delay=initial_delay)
12   p = new Printer()
13   c.tick -> p.tick
14 }
15
16 reactor Clock(
17   step: time = 0,
18   initial_delay:time = 0
19 ) {
20   output tick: void
21   logical action a
22
23   state delay: time = initial_delay
24
25   reaction(startup a) -> a, tick {=
26     a.schedule(delay);
27     tick.set();
28     delay += step;
29   =}
30 }
```



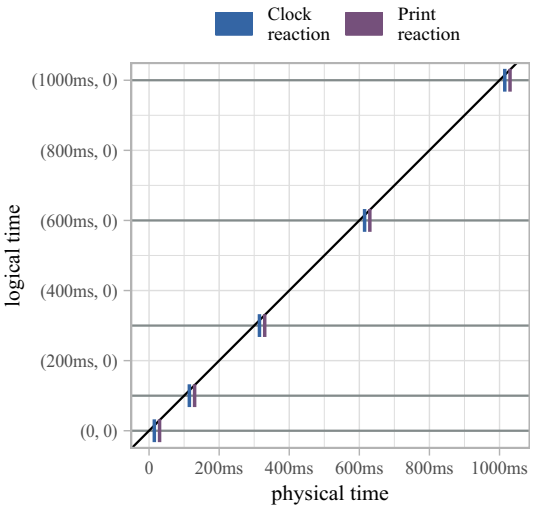


Figure 4.6: Timing diagram for the slowing clock program in Listing 4.9.

When executed, the program prints:

```
Tick at 0 nsecs
Tick at 100000000 nsecs
Tick at 300000000 nsecs
Tick at 600000000 nsecs
Tick at 1000000000 nsecs
...
```

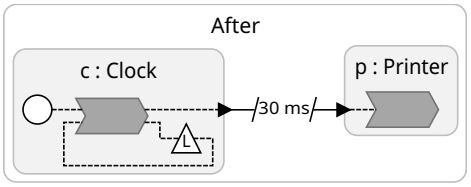
Figure 4.6 shows the corresponding timing diagram. The interval between events is increasing with each invocation of the clock reaction.

4.4.5 After Delays

In the reactor model, relaying messages via connections is logically instantaneous. However, the LF syntax optionally allows for the specification of a logical delay on a connection using the **`after`**keyword. This is an extension of the underlying reactor model. Consider, for example, the program in Listing 4.10. It replicates the example in Listing 4.8 but connects the `Clock` and `Print`reactors using an after delay of 30 ms.

Listing 4.10: Example LF program illustrating the use of after delays.

```
target Cpp

import Clock, Printer from "SimpleClock.lf"

main reactor {
  c = new Clock(period = 100 ms)
  p = new Printer()
  c.tick -> p.tick after 30 ms
}
```



When executed, the program prints:

```
Tick at 30000000 nsecs
Tick at 130000000 nsecs
Tick at 230000000 nsecs
Tick at 330000000 nsecs
Tick at 430000000 nsecs
...
```

The corresponding timing diagram is given in Figure 4.7. When the clock reaction sets the output port, it schedules a new event with an offset of 30 ms. This new event triggers the print reaction.

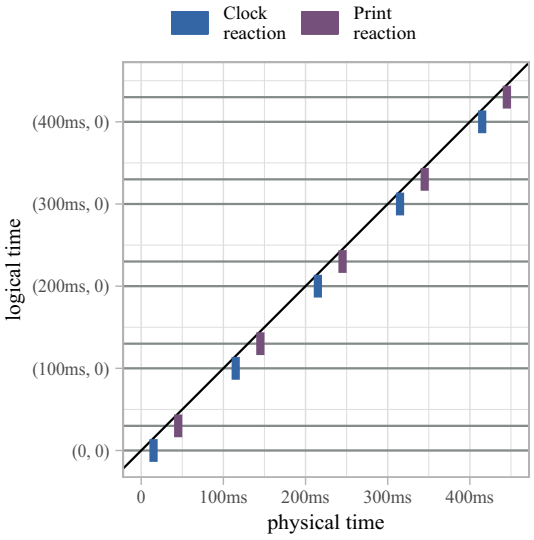
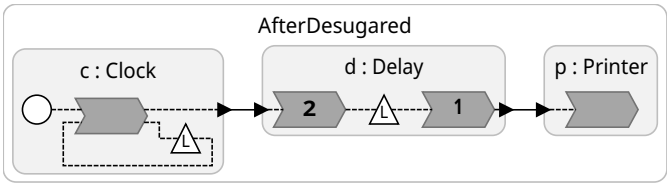


Figure 4.7: Timing diagram for the after delay example in Listing 4.10.

```
target Cpp

import Clock from "SimpleClock.lf"
import Printer from "SimpleClock.lf"

reactor Delay(delay: time = 0) {
  input in: void
  output out: void
  logical action a: void

  reaction(a) -> out {=
    out.set();
  =}

  reaction(in) -> a {=
    a.schedule(delay);
  =}
}

main reactor{
  c = new Clock(period = 100ms)
  d = new Delay(delay = 30ms)
  p = new Printer()
  c.tick -> d.in
  d.out -> p.tick
}
```

Listing 4.11: Example LF program that is semantically equivalent to the after delay example in Listing 4.10 but uses an explicit delay reactor.



After delays are semantically equivalent to using a delay reactor like the one used in Listing 4.5. Listing 4.11 shows a modified version of the after delay example in Listing 4.10. This program uses a delay reactor with a logical action instead of the after delay syntax that LF provides. Since after delays can be implemented using plain reactor mechanisms, the after delays in LF can be considered syntactic sugar. In fact, the LF compiler implements a transformation that replaces after delays with a synthesized delay reactor. This is used for targets that do not provide native support for after delays in their runtime.

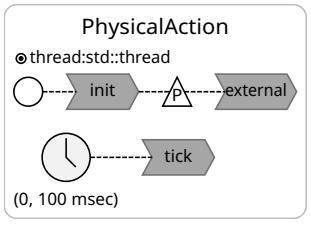
4.4.6 Physical Actions

Unlike logical actions, which are scheduled relative to the current tag, physical actions are scheduled relative to the current reading of physical time (cf. Section 3.1.2). This makes physical actions particularly useful for scheduling events in an external context outside the scope of the LF program. This could be external concurrent processes or interrupts that are triggered by some external event.

Physical actions provide a means for asynchronous processes to safely insert an event into the event queue; the scheduling of a physical action can be

Listing 4.12: Example LF program illustrating the use of physical actions.

```
target Cpp

main reactor{
  timer t(0, 100 ms)
  physical action p: void

  state thread: std::thread

  reaction init(startup) -> p {=
    // Start an asynchronous thread
    // that waits for roughly 250 ms
    // and then schedules the
    // physical action.
    thread = std::thread([&] () {
      std::this_thread::sleep_for(250ms);
      p.schedule();
    });
  =}

  reaction tick(t) {=
    std::cout
      << "React to timer tick at "
      << get_elapsed_logical_time()
      << '\n';
  =}

  reaction external(p) {=
    std::cout
      << "React to external event at "
      << get_elapsed_logical_time()
      << '\n';
  =}
}
```



seen as providing an input to the system. The tag assigned to the event is nondeterministic in the sense that it is not defined by the program. Therefore, the tag is considered to be part of the input. Once the tag is assigned, however, the scheduled event is inserted into the event queue and processed deterministically and strictly in tag order.

Listing 4.12 provides an example program that illustrates the use of physical actions in LF. The main reactor contains a timer `t` and the physical action `p`, as well as the state variable **`thread`** which holds a thread object. The program spawns this thread at startup in the `init` reaction on line 14. The thread models an asynchronous external process that may schedule the physical action. Concretely, the thread waits for roughly 250 ms and then schedules the physical action once before it terminates.

The main reactor further defines the reaction `tick`, which is triggered by the timer `t`, and the reaction `external`, which is triggered by the externally scheduled physical action `p`. Both reactions print their trigger and the current logical time. When executing the program, it prints something similar to the following:

```
1 React to timer tick at 0 nsecs
2 React to timer tick at 100000000 nsecs
3 React to timer tick at 200000000 nsecs
4 React to external event at 250556992 nsecs
5 React to timer tick at 300000000 nsecs
6 React to timer tick at 400000000 nsecs
7 ...
```

The precise tag assigned to the external event may vary between executions.

The corresponding timing diagram is shown in Figure 4.8. In the diagram, we assume that `schedule` is called at physical time 255 ms. This physical time is marked by the vertical dashed brown line. The newly scheduled event is inserted at tag (255 ms 0), and this event is handled promptly by triggering and executing the `external` reaction.

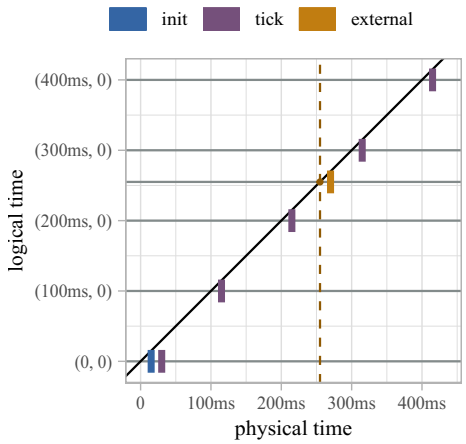
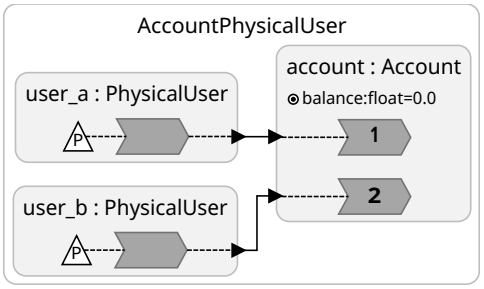


Figure 4.8: Timing diagram for the physical action example in Listing 4.12.

We can use a pattern similar to Listing 4.12 to model external asynchronous input. Consider again the bank account example shown in Listing 4.3 and discussed in Section 4.3.1. In this example, the order in which the users send requests is hard-coded using timers, and thus the requests have predetermined fixed tags. While using a predefined order is useful for testing and demonstration, reactor programs that are deployed in practice need to be able to handle sporadic asynchronous inputs to be useful. Physical actions provide the means to achieve this.

Listing 4.13: A variant of the bank account example in Listing 4.3 using physical actions to model the user input.

```
target Cpp

import Account from "SimpleAccount.lf"

reactor PhysicalUser{
  output request: float
  physical action a: float

  reaction(a) -> request {=
    request.set(std::move(a.get()));
  =}
}

main reactor{
  account = new Account()
  a = new PhysicalUser()
  b = new PhysicalUser()

  user_a.request -> account.request_a
  user_b.request -> account.request_b
}
```



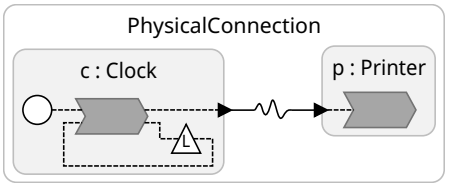
Listing 4.13 shows a modified version of the simple account example that uses physical actions to model sporadic user input. Note that the scheduling of the physical actions is not part of the program and needs to be done externally. The `Account`reactor remains unchanged. Consequently, it implements the same testable behavior as in the earlier examples. Concretely, if `user_a` sends a deposit message at tag g_a and `user_b`sends a withdrawal message at tag g_b with $g_b > g_a$, then the semantics of LF guarantees that the response of the account is identical to the response in a test case that uses the same tag ordering (i.e., the behavior is identical to the program in Listing 4.3). Physical actions draw a clear perimeter around the deterministic and therefore testable program logic while allowing it to interact with sporadic external inputs.

4.4.7 Physical Connections

The LF syntax also provides a mechanism for inserting physical connections. A physical connection can be created using the `~>` operator instead of the regular `->` connection operator. Similar to connections with after delays, physical connections can be replaced with a reactor that internally uses a physical action to delay the message. Consider the example program in Listing 4.14, which uses a physical connection to connect the `Clock`and `Print` reactors. It is semantically equivalent to the program given in Listing 4.15.

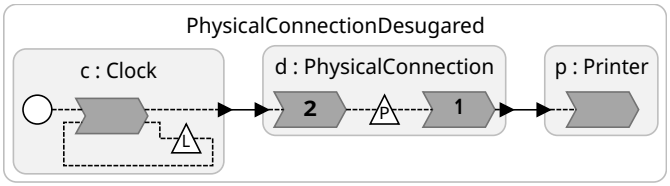
Listing 4.14: Example LF program illustrating the use of physical connections.

```
target Cpp

import Clock, Printer from "SimpleClock.lf"

main reactor{
  c = new Clock(period = 100ms)
  p = new Printer()
  c.tick ~> p.tick
}
```



```
target Cpp

import Clock from "SimpleClock.lf"
import Printer from "SimpleClock.lf"

reactor PhysicalConnection {
  input in: void
  output out: void
  physical action a: void

  reaction(a) -> out {=
    out.set();
  =}

  reaction(in) -> a {=
    a.schedule();
  =}
}

main reactor {
  c = new Clock(period = 100ms)
  d = new PhysicalConnection()
  p = new Printer()
  c.tick -> d.in
  d.out -> p.tick
}
```

Listing 4.15: Example LF program that is semantically equivalent to the physical connection example in Listing 4.14, but that uses an explicit physical delay reactor.



When executed, the program prints something similar to the following:

```
Tick at 389958 nsecs
Tick at 100079622 nsecs
Tick at 200086711 nsecs
Tick at 300080375 nsecs
...
```

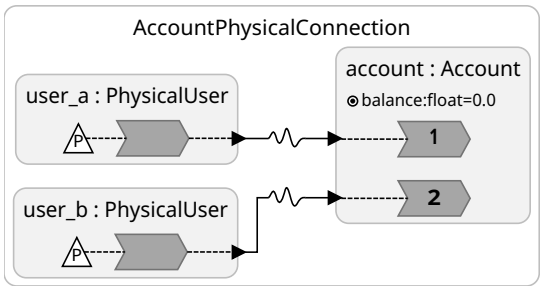
The precise times printed may vary between executions.

In this example, the physical action is not scheduled from an external concurrent process but from inside the LF program. This is useful for deliberately introducing actor-like nondeterminism into a program when this is desired. By using a physical connection, the programmer explicitly states that the receiver should ignore the logical ordering of messages relative to other events in the system and instead process messages in the order of arrival.

Using both physical actions and physical connections, we can replicate the semantics of actor programs in LF. Consider the modified bank account example in Listing 4.16 that uses physical connections to connect the users with the account. In this example, the deposit and withdrawal messages are tagged and processed nondeterministically in the order in which they arrive at the account. This behavior is similar to the nondeterministic behavior of the actor implementation shown in Figure 2.6a on Page 16.

Listing 4.16: A nondeterministic implementation of the bank account example in Listing 4.3 using physical actions to model the user input and physical connections to relay the user messages.

```
target Cpp

import Account from "SimpleAccount.lf"
import PhysicalUser from "AccountPhysicalUser.lf"

main reactor {
  account = new Account()
  user_a = new PhysicalUser()
  user_b = new PhysicalUser()

  user_a.request ~> account.request_a
  user_b.request ~> account.request_b
}
```



4.4.8 Reflex Game

The example program in Listing 4.17 illustrates how both logical and physical actions can be used to implement an interactive application in LF. The program implements the Reflex Game, which is inspired by a similar Esterel example program.[11] This simple game measures the player's reaction time. The game prints a prompt at a random time and the player needs to react by pressing the enter or return key. The game measures the time that elapsed between the prompt and the player's reaction. If the player presses enter before the prompt appears, the game considers this cheating and quits.

11: Géard Berry and Gonthier 1992, *The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation*.

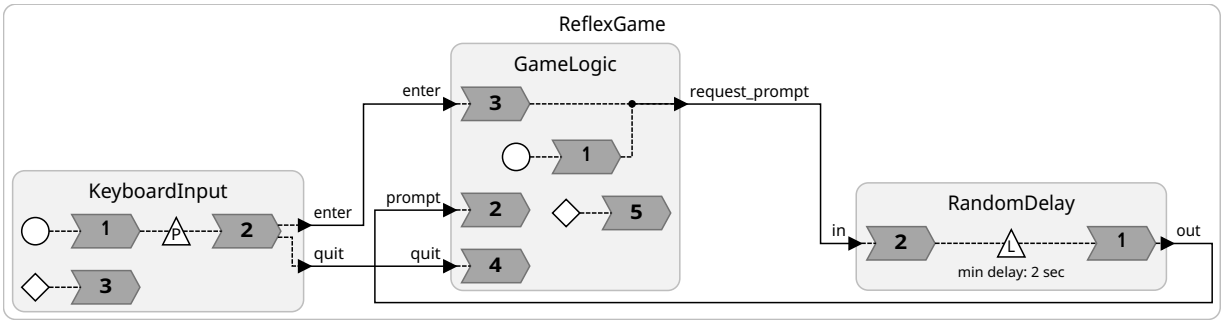


Figure 4.9: Diagram showing the Reflex Game defined in Listing 4.17.

Figure 4.9 shows the reactor diagram for the Reflex Game. The program consists of three reactors: `KeyboardInput`, `RandomDelay`, and `GameLogic`. The `KeyboardInput` reactor interacts with the keyboard and informs the game logic when a key was pressed. Similar to the example in Listing 4.12, it spawns an additional thread that executes asynchronously and uses a physical action to notify the `KeyboardInput` reactor when a key is pressed. The `getchar()` function, which is used to read from the keyboard, blocks until a key is pressed. Calling this function directly from a reaction could block the entire program. Therefore, the additional thread is needed to enable blocking calls while ensuring that the reactor program can progress independently.

The `RandomDelay` reactor is similar to the delay reactor in Listing 4.11; it forwards the input signal to the output with a logical delay. However, the delay is not fixed but determined by a random number generator. For each message received on the input port, the reaction triggered by this message determines a random delay between 2 s and 8 s and schedules its internal logical action accordingly.

Finally, the `GameLogic` reactor implements the core logic of the game. It prints usage instructions on startup and then requests a prompt by sending a message to `RandomDelay`. Once the message arrives back at the `GameLogic` reactor, it displays the prompt and waits for the player's reaction, which is indicated by a message from `KeyboardInput` on the `enter` port. The game calculates and prints the reaction time and then requests another prompt. If the player presses enter too early or quits the game, the program terminates.

This example illustrates the principle usage of LF for implementing an application that requires reacting to external input, implementing some predictable behavior, and performing actions at certain points in time. Admittedly, using an additional thread for interacting with the keyboard is inconvenient, as it requires breaking out of the reactor model. Chapter 6 discusses a better solution that allows some reactions to perform long-running computations or call blocking functions while the rest of the program can progress independently.

Listing 4.17: Reflex Game implemented in LF.

```
target Cpp

reactor RandomDelay{
  public preamble{=
    #include <random>
  =}

  input in: void
  output out: void
  // logical action with a minimum delay of 2s
  logical action delay(2s): void
  // a random number generator seeded with the
  // current physiceal time
  state rand: std::mt19937({=
    get_physical_time().time_since_epoch().count()
  =})
  // a uniform random distribution with a range
  // from 0 to 6,000 milliseconds
  state dist: std::uniform_int_distribution<int>(
    0, 6000
  )

  reaction(delay) -> out {= out.set(); =}

  reaction(in) -> delay {=
    delay.schedule(dist(rand) * 1ms);
  =}
}

reactor KeyboardInput{
  state thread: std::thread
  state terminate: std::atomic<bool> = false
  physical action keyboard_input: char

  output enter: void
  output quit: void

  reaction(startup) -> keyboard_input {=
    // start an external thread that listens for
    // keyboard input.
    thread = std::thread([&] () {
      while(!terminate.load()) {
        int c = getchar();
        keyboard_input.schedule(c);
      }
    });
  =}

  reaction(keyboard_input) -> enter, quit {=
    char key = *keyboard_input.get();
    if(key == '\n') {
      enter.set();
    } else if (key == EOF) {
      quit.set();
    }
  =}

  reaction(shutdown) {=
    terminate.store(true);
    thread.join();
  =}
}

reactor GameLogic{
  private preamble{=
    using namespace std::chrono;
  =}

  output request_prompt: void
  input prompt: void
  input enter: void
  input quit: void

  state prompt_time: {= reactor::TimePoint =} =
    {= reactor::TimePoint::min() =}
  state total_time: time(0)
  state count: unsigned = 0

  reaction(startup) -> request_prompt {=
    std::cout
      << "********************************\n"
      << "Hit Return or Enter when prompted.\n"
      << "Type Control-D (EOF) to quit\n\n";

    request_prompt.set(); // request the first prompt
  =}

  reaction(prompt) {=
    prompt_time = get_physical_time();
    std::cout <<"\nHit Return or Enter\n";
  =}

  reaction(enter) -> request_prompt {=
    // If the prompt_time is min(), then the user
    // is hitting return before being prompted.
    if (prompt_time == reactor::TimePoint::min()) {
      std::cout <<"YOU CHEATED\n";
      environment()->sync_shutdown();
    } else {
      reactor::TimePoint logical = get_logical_time();
      auto elapsed = (logical - prompt_time);
      auto time_in_ms =
        duration_cast<milliseconds>(elapsed);
      std::cout <<"Response time in milliseconds: "
                << time_in_ms <<'\n';
      count++;
      total_time += time_in_ms;
      // Reset the prompt_time to indicate that there
      // is currently no prompt shown.
      prompt_time = reactor::TimePoint::min();
      // Request another prompt.
      request_prompt.set();
    }
  =}

  reaction(quit) {= environment()->sync_shutdown(); =}

  reaction(shutdown) {=
    if (count > 0) {
      auto avg = total_time / count;
      std::cout <<"\n**** Average response time: "
        << duration_cast<milliseconds>(avg) <<'\n';
    } else {
      std::cout <<"\n**** No attempts\n";
    }
  =}
}

main reactor{
  delay = new RandomDelay()
  keyboard = new KeyboardInput()
  logic = new GameLogic()

  logic.request_prompt -> delay.in
  delay.out -> logic.prompt
  keyboard.enter -> logic.enter
  keyboard.quit -> logic.quit
}
```

4.5 Federated Execution: Coordination Across Multiple Timelines

Lingua Franca also supports the generation of programs that may be executed in a distributed system. Any LF program can be converted into a distributed (or federated) program by exchanging the `main` modifier of the top-level reactor for the `federated` modifier.[12] If the `federated` modifier is used, then each reactor instance created within the federated reactor constitutes a *federate*. Each federate is compiled as an individual reactor program with its own executable. We refer to the complete program, consisting of all federates, as a *federation*.

12: Currently, federated programs are only fully supported by the C and Python targets.

The support for federated execution of LF programs is not a primary contribution of this thesis. The key authors of the methodology and implementation for federated execution are Edward A. Lee, Marten Lohstroh and Soroush Bateni. This chapter includes a discussion of federated execution to provide a complete overview of Lingua Franca.

4.5.1 Aircraft Door Example

To illustrate the programming and execution of federated LF programs, we consider again the example of an aircraft door first discussed in Section 2.3.2. This example considers an aircraft equipped with emergency slides. To safely open the passenger door in the parking position, the pilots in the cockpit first need to disarm the slide and then open the door. The example assumes that the disarm message from the cockpit is routed through an additional sensor component that performs safety checks. Figure 4.10 shows again the Hewitt actor implementation of this example.

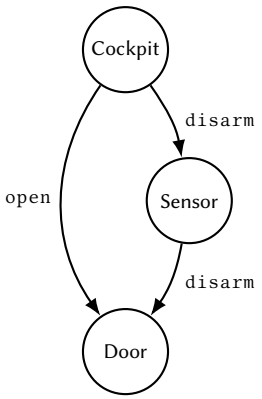


Figure 4.10: An actor implementation of the aircraft door example. (Repeated from Figure 2.7)

As Section 2.3.2 discusses, we cannot make any assumptions about the order in which the door actor processes the incoming messages. Since the `disarm` message is relayed via the sensor actor, there is no guarantee that the `open` and `disarm` messages arrive at `Door` in the order in which they were sent. Figure 4.11 shows a possible message exchange between the actors. Although `Cockpit` sends first the `disarm` message and then the `open` message, `open` arrives before `disarm` at `Door`, and `Door` will perform the corresponding actions in this order. Thus, the Hewitt actor implementation does not provide a safe solution to the problem.[13]

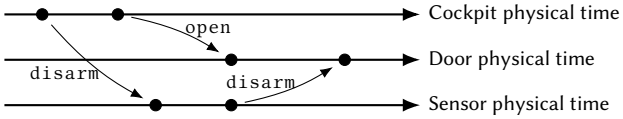


Figure 4.11: Different actors may observe events in a different order.

13: We use Hewitt actors here as a placeholder that represents a whole family of possible solutions that also includes services and publish/subscribe, which build on similar principles as actors

There are various ways in which the actor program can be improved. For instance, `Cockpit` could send another message directly to `Door` that indicates that a disarm message was sent to `Sensor` and that `Door` needs to wait for a message from `Sensor` before opening the door. Alternatively, Sensor could send a message back to Cockpit, and Cockpit would then send the `disarm` message directly to the Door. However, any of the possible solutions adds more complexity, and reasoning about the correctness of the problem becomes harder to reason about when there are more actors involved.

The semantics of Lingua Franca instead guarantees that Door processes the incoming messages in the logical order in which they were sent. The federated flavor of LF extends this guarantee to a distributed context and

automatically provides an implementation that remains faithful to the program specification. This relieves the application designer from developing complex protocols and reasoning about the possible interleaving of messages. Figure 4.12 shows the diagram of an LF realization of the aircraft door example.

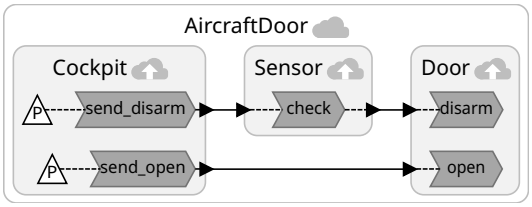


Figure 4.12: Federated LF implementation of the aircraft door example.

In federated execution, each federate keeps track of its own logical timeline. Each federate is responsible for processing its local events and deciding when to advance to the next tag. To do this deterministically, however, we need to annotate messages sent over network connections with their tag, and we need a coordination scheme that allows each federate to decide when it is safe to advance to the next tag. A federate may only advance its logical time when it knows that no other federate can send a message with an earlier tag. To achieve this, Lingua Franca currently implements two coordination strategies: *centralized coordination* and *decentralized coordination*, which we discuss in the following.

4.5.2 Centralized Coordination

In the centralized coordination scheme, all federates communicate with a central coordinator, which is called the run-time infrastructure (RTI). This approach is closely related to the High Level Architecture (HLA) standard, which also uses similar terminology.[14] Figure 4.13 visualizes the overall architecture.

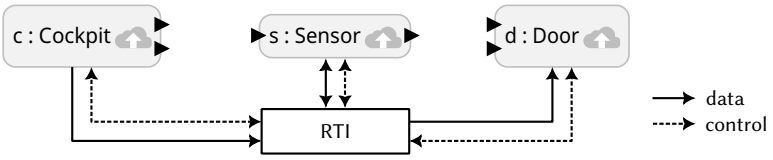


Figure 4.13: The centralized coordination scheme for federated execution of LF programs.

14: Dahmann, Fujimoto, and Weatherly 1997, *The Department of Defense High Level Architecture*; Kuhl, Weatherly, and Dahmann 1999, *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*.

In centralized coordination, the federates do not communicate directly with each other. All tagged messages on connections between federates are routed via the central RTI. In addition to the data messages, the federates exchange control messages with the RTI. Each federate is responsible for informing the RTI about the next tag at which it may send a message to another federate. The RTI, in turn, is responsible for coordinating the advancement of the logical time of the federates. It sends grants which inform a federate that it may safely advance execution until a certain tag.

While this approach guarantees the correct federated execution of LF programs, it comes with several disadvantages. Since all messages are routed via the RTI, the RTI becomes both a single point of failure and a potential performance bottleneck. Also, the centralized coordination scheme requires frequently sending null messages if a federate contains a physical action that can trigger an outgoing network message. In the presence of physical actions, we cannot infer the earliest next tag at which the federate may produce an event. Therefore, it needs to inform the RTI about the presence or absence of events at regular intervals.

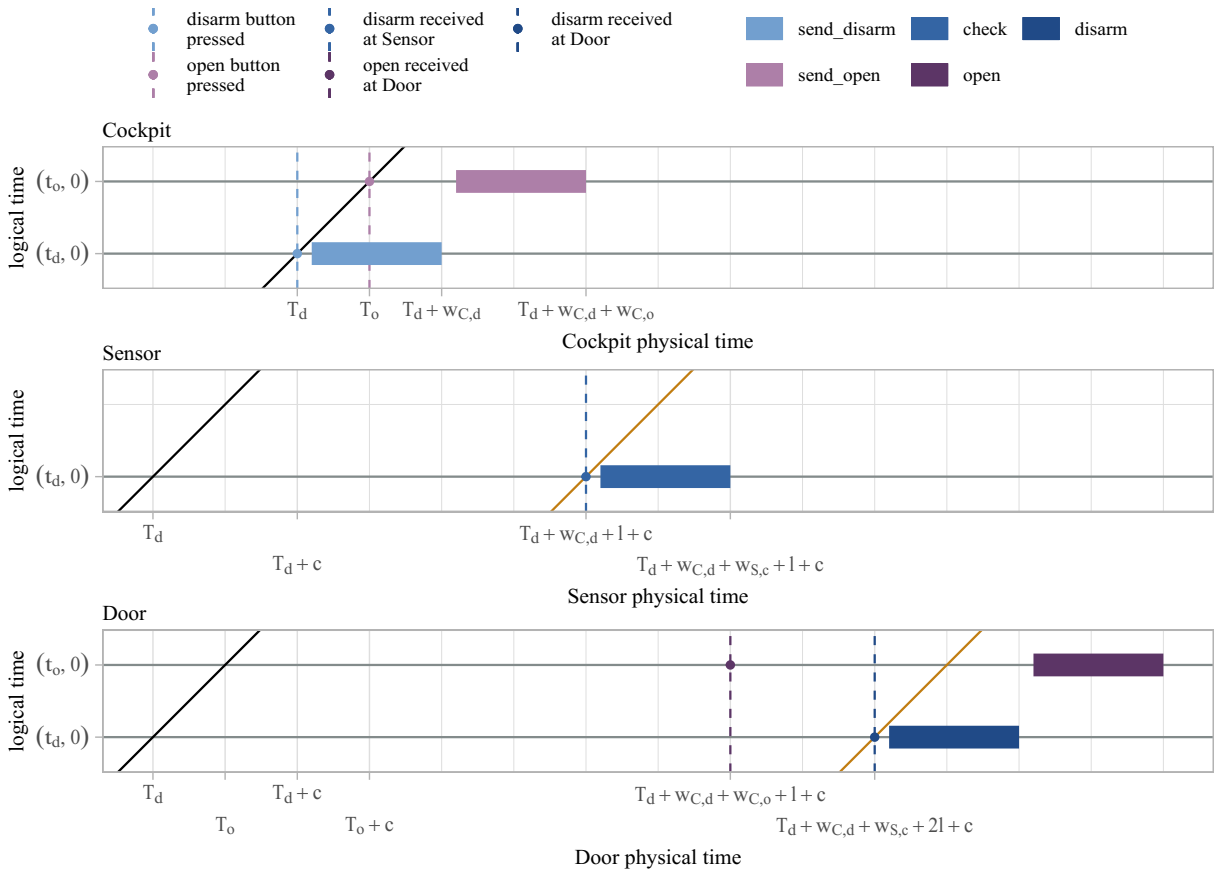


Figure 4.14: Timing diagrams for the aircraft door example that illustrate the decentralized coordination scheme for federated execution of LF programs.

4.5.3 Decentralized Coordination

In the decentralized coordination scheme, the federates communicate directly with each other. To decide when it is safe to advance logical time, the federates utilize the safe-to-process analysis known from PTIDES.[15] This is the same technique as the one used by DEAR for coordinating reactors on top of AUTOSAR AP (cf. Section 3.3.2).

15: Y. Zhao, J. Liu, and E. A. Lee 2007, *A Programming Model for Time-Synchronized Distributed Real-Time Systems*; Derler, Feng, et al. 2008, *PTIDES: A Programming Model for Distributed Real-Time Embedded Systems*.

By making assumptions about the maximum clock synchronization error c, the network latency l and the WCET w of all downstream reactions that may send a network message, each federate can compute a safe-to-process offset s, which allows the federate to infer when it is safe to advance to the next tag.

Consider the timing diagrams in Figure 4.14. They show a fictive execution trace for the aircraft door example. In this trace, the pilot presses the disarm button at the physical time T_d, and shortly after they press the open button at T_o. Consequently, the two events scheduled via the physical actions in the `Cockpit`reactor get assigned tag $(t_d, 0)$ with $t_d = T_d$ for the `disarm`event and tag $(t_o, 0)$ with $t_o = T_o$ for the `open`event.

To keep the example and the discussion simple, we assume that the worst-case delays and the clock synchronization error have identical values. Specifically, $w_{C,d}$ denotes the WCET of the `send_disarm`reaction of Cockpit, $w_{C,o}$ denotes the WCET of the `send_open`reaction of Cockpit, $w_{S,c}$ denotes the WCET of the `check`reaction of Sensor, and we assume:

$$w_{C,d} = w_{C,o} = w_{S,c} = l = c\,.$$

The runtime scheduler executes the `send_disarm` reaction of `Cockpit` immediately after the corresponding event is scheduled. In the shown execution trace, the processing of the reaction completes exactly at $T_d + w_{C,d}$ and `Cockpit` sends the disarm network message to Sensor precisely at this instant. `Cockpit`'s scheduler then advances the current tag to $(t_o, 0)$ and starts processing the `open` event by executing the `send_open` reaction. Since there is an overlap with the execution of `send_disarm`, the `send_open` reaction is executed with a slight lag. It completes its computation at $T_d + w_{C,d} + w_{C,o}$ and sends the `open` network message to `Door` precisely at this instant.

From `Cockpit`'s point of view, the `disarm` message arrives at `Sensor` at $T_d + w_{C,d} + l$. However, the local physical clock of `Sensor` has a skew of c relative to the `Cockpit`'s clock. Therefore, the x-axis of the `Sensor` plot in Figure 4.14 is offset by c to the left. Therefore, `Sensor` receives the `disarm` message at $T_d + w_{C,d} + l + c$ according to its local clock. According to our assumptions, this point in time also marks the latest point at which `Sensor` may receive a message from `Cockpit` with the tag $(t_d, 0)$. Consequently, `Sensor` has the safe-to-process offset $s_S = w_{C,d} + l + c$. In the diagram, this safe-to-process offset is indicated by the orange line. Due to this offset, `Sensor` may only process events with a timestamp t with $t \geq T - s_S$, where T is the current reading of physical time.

`Sensor` forwards the `disarm` message to `Door` precisely at $T_d + w_{C,d} + w_{S,c} + l + c$ and `Door` receives the message at $T_d + w_{C,d} + w_{S,c} + 2l + c$ according to its local clock. Note that the maximum clock synchronization error c indicates the maximum clock skew between all federates. Thus, we only have to account for it once, even if there are multiple network hops along the chain of events.

The `open` message from `Cockpit`, however, arrives at $T_d + w_{C,d} + w_{C,o} + l + c$ according to the local clock of `Door`. This is before the `disarm` message from `Sensor` arrives. Consequently, the local scheduler needs to wait before processing the `open` message, until it can be sure that there is no pending `disarm` message with an earlier tag. The maximum delay for the arrival of the `disarm` message relayed via `Sensor` is $w_{C,d} + w_{S,c} + 2l + c$. The maximum delay for arrival of the `open` message is $w_{C,d} + w_{C,o} + l + c$. Note that we need to account for the WCET of both the `send_disarm` and the `send_open` reactions, as in the worst-case scenario both the `open` and the `disarm` events are logically simultaneous, and both reactions execute in sequence. The safe-to-process offset s_D that `Door` needs to account for is given as the maximum of both delays:

$$s_D = \max\left\{w_{C,d} + w_{S,c} + 2l + c, w_{C,d} + w_{C,o} + l + c\right\}.$$

For the values chosen in this example, the worst-case delay of the disarm message is larger, and hence we have $s_D = w_{C,d} + w_{S,c} + 2l + c$. This safe-to-process offset is also indicated by the orange line in the bottom plot in Figure 4.14.

The safe-to-process offset provides an additional barrier, which tells each federate precisely how long it should wait until it can be sure that is has seen all earlier messages. If a message with an earlier tag arrives although the federate has waited until the safe-to-process barrier, then this indicates that one or multiple of the assumptions have been violated. This can be handled as a runtime fault by the program logic.

The decentralized coordination eliminates the bottleneck and single point of failure that the RTI presents in centralized coordination. However, it requires WCET analysis and known bounds on latency and clock synchronization errors. By using time-predictable hardware and a real-time OS, the bounds

on execution time can be tightened. Additionally, techniques like Time-Sensitive Networking (TSN) can be used to tighten the bounds on network latency,[16] and Precision Time Protocol (PTP) can be used to minimize the clock synchronization error.[17]

16: Finn 2018, *Introduction To Time-Sensitive Networking*; Austad and Mathisen 2023, *Bounding the End-to-End Execution Time in Distributed Real-Time Systems: Arguing the Case for Deterministic Networks in Lingua Franca*.

17: IEEE 2019, *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*.

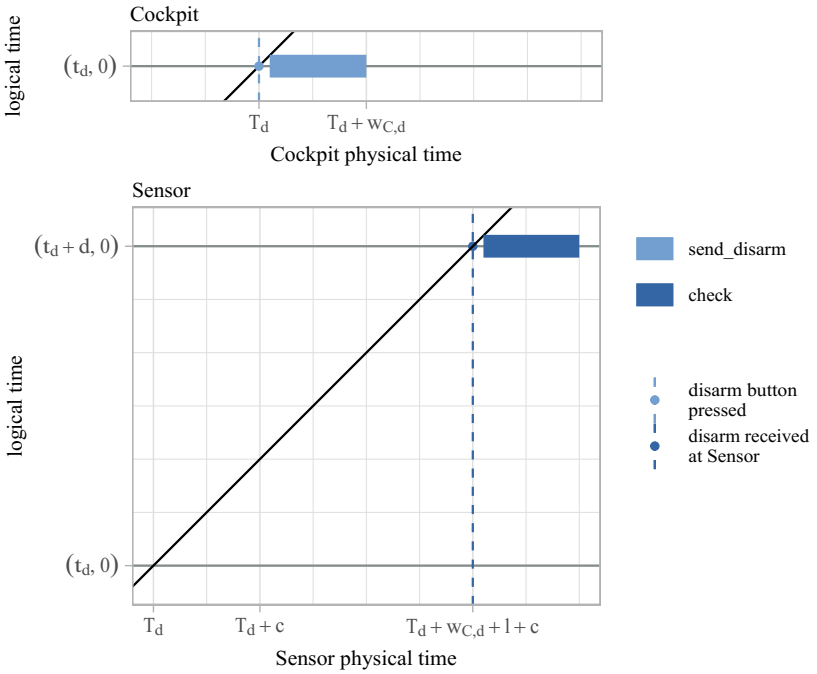


Figure 4.15: Decentralized coordination with a logical delay $d = w_{C,d} + l + c$ on the connection between Cockpit and Sensor.

It is also possible to add an after delay d on a connection between federates. This logical delay can be subtracted from the safe-to-process offset. For instance, the safe-to-process offset for `Sensor`becomes $s_S = w_{C,d} + l + c - d$, for any non-zero delay. In the special case of choosing $d = w_{C,d} + l + c$ the safe-to process offset becomes zero. This special case is illustrated in the timing diagram in Figure 4.15. It is equivalent to the distributed coordination for DEAR discussed in 3.3.2 and to (system-level) LET. However, the decentralized coordination as implemented in LF is more general, as it can handle the full spectrum between no logical delay as in the synchronous-reactive paradigm and a logical delay that accounts for the worst-case end-to-end latency as in LET.[18]

18: E. A. Lee and Lohstroh 2022, *Generalizing Logical Execution Time*.

4.5.4 Trading off Consistency and Availability

Eric Brewer formulated the well-known CAP theorem, which highlights the fundamental limitations of distributed systems.

> The CAP theorem states that any networked shared-data system can have at most two of three desirable properties:
>
> - *consistency* (C) equivalent to having a single up-to-date copy of the data;
> - *high availability* (A) of that data (for updates); and
> - tolerance to network partitions (P).[19]

19: Brewer 2012, *CAP Twelve Years Later: How the "Rules" Have Changed*.

This asserts that when a system becomes partitioned due to increasing network delays, then either availability or consistency need to be sacrificed.

In Lingua Franca, the centralized coordination favors consistency over availability. The RTI grants a tag advancement only once all earlier messages have arrived at the federate. In case of network failure, the federate might wait

indefinitely. The decentralized coordination, however, favors availability over consistency. When network latencies increase beyond the assumed worst-case latency l, the federates locally decide to advance their current tag, although there might be earlier messages that have not arrived yet. If a message with an earlier tag arrives late, this can be detected and handled by user code similarly to deadline violations.

Brewer also suggests that availability and consistency are not fully exclusive in the case of a partitioned system. There is a trade-off between achieving high availability and achieving consistency that Edward A. Lee quantified in the CAL theorem.[20] This trade-off is exposed by Lingua Franca as users may freely choose the logical delay imposed on a connection.

20: E. A. Lee, Bateni, et al. 2021, *Quantifying and Generalizing the CAP Theorem*; E. A. Lee, Bateni, et al. 2023, *Trading Off Consistency and Availability in Tiered Heterogeneous Distributed Systems*; E. A. Lee, Akella, et al. 2023, *Consistency Vs. Availability in Distributed Cyber-Physical Systems*.

In the CAL theorem, consistency is defined in terms of logical delay. Consider, for instance, a sensor reactor that frequently sends data to a control reactor. If there is no delay on the connection, then the sensor and the controller reaction are executed logically simultaneously and both agree on the current sensor value. If, however, there is a logical delay between the two reactors, then the sensor could already have sent an updated value when the controller reads the first value. Thus, they have an inconsistent view of the system's state.

The CAL theorem further defines availability in terms of deadlines in LF. In this definition, availability is an upper bound on the lag between the physical timeline of execution and the logical timeline as given by the tags. Thus, we can specify an upper bound on availability using deadlines, and we can use logical delays to deliberately trade consistency for availability. This is highlighted in Figure 4.15, where an after delay d is used to increase availability by reducing the safe-to-process offset to zero and thus eliminating potential waiting time. However, this also decreases consistency as Sensor's view of the system state corresponds to a logical tag that is in the past.

4.6 The Lingua Franca Toolchain

A Lingua Franca program is a mixture of LF code and target code. To make programming in such a hybrid setting a fluent experience, adequate tooling is key. LF comes with a set of standalone command-line tools, as well as support for different integrated development environments (IDEs). Currently, there is an Eclipse-based IDE named Epoch and a Visual Studio Code extension. All LF tools and IDEs share a common code base and provide a similar set of features. Most notable is the use of interactive diagrams to facilitate the development of LF programs. Figure 4.16 provides an overview of the Lingua Franca toolchain.

4.6.1 Compilation

The backbone of the LF compiler is the Xtext framework, which applies a model-based approach to creating domain-specific languages.[21] Based on a grammar definition, Xtext automatically generates the basic compiler infrastructure as well as an Eclipse plugin and a language server implementing the Language Server Protocol (LSP).[22] The Xtext-based workflow can be broken down into lexical analysis, parsing, validation, and code generation, where lexical analysis and parsing are completely handled by Xtext. Xtext automatically generates a metamodel based on the Eclipse Modeling Framework (EMF)[23] and provides an abstract syntax tree (AST) in the form of a concrete model for each successfully parsed program.

21: Eysholdt and Behrens 2010, *Xtext: Implement Your Language Faster than the Quick and Dirty Way*.

22: Bünder 2019, *Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages*.

23: Steinberg et al. 2008, *EMF: Eclipse Modeling Framework*.

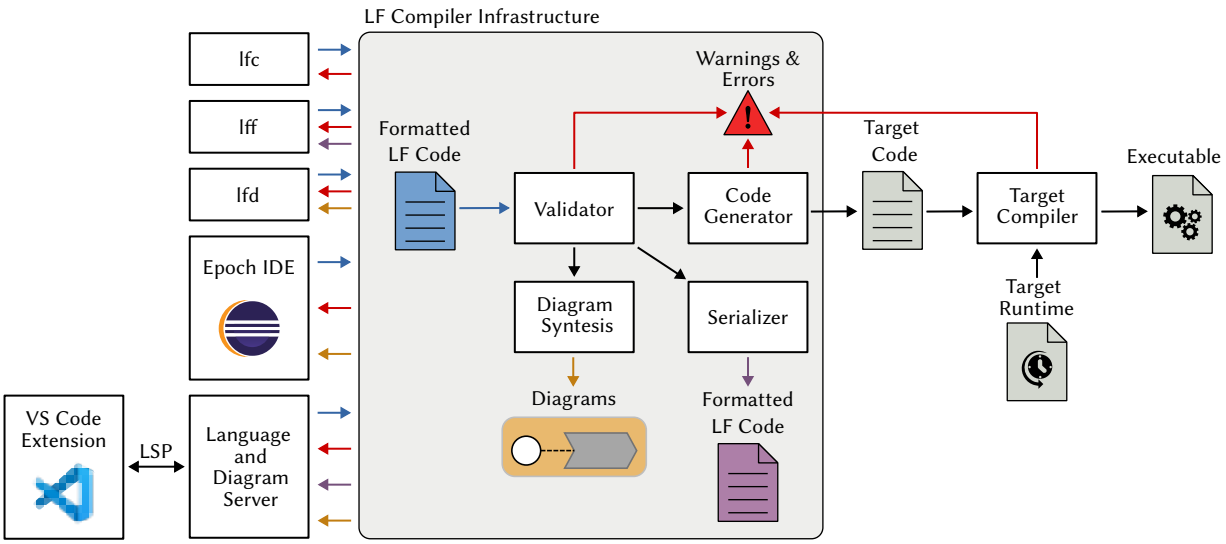


Figure 4.16: The Lingua Franca toolchain.

Once the model representing the program is created, the LF validator performs various semantic checks. This includes simple assertions such as the existence of at most one main (or federated) reactor, whether types declarations are present (in case the target language is statically typed), or whether certain language features are actually supported by the specified target. The validator also performs structural analysis on the composition of reactors to ensure that there are no semantic problems like cyclic instantiations or causality loops (i.e., closed loops of dependent reactions without any delays). The target code blocks in LF programs are treated as a black boxes, and error-checking on them is deferred to the target toolchain.

After validation, a code generator maps the EMF model to constructs in the selected target language. The generated code is then compiled using the target language's toolchain (unless the target is an interpreted language). Since the LF compiler does not check the embedded target code blocks, it relies on the target compiler to report any issues within those blocks. Similarly, any type-checking, e.g., on the ports of a connection, is deferred to the target compiles. If compilation fails, then the error messages are collected and mapped back to the appropriate locations in the original LF code. Similar to the errors reported during the validation phase, error messages obtained from the target compiler are presented to the user either in the command line output or through the IDE user interface.

4.6.2 Code Generators and Runtime Implementations

Lingua Franca is designed to intermix with arbitrary target languages. This polyglot nature of LF enables an interesting symbiosis between the reactor-based coordination model and the unique capabilities provided by the target language. A wide range of supported targets not only accommodates programmers who are used to certain programming styles but also brings features from the target languages to LF, enabling the application of LF to more domains.

As of this writing, LF supports C, C++, Rust, Python, and TypeScript as target languages. Each target consists of a code-generation backend specifically designed for the target language, as well as a runtime environment that supports the execution of LF programs. While the code generators are part

of the main compiler infrastructure,[24] the target runtimes are developed separately and independently (to a varying degree).[25]

There are multiple trade-offs involved in the design of an LF backend. Each of the existing runtime implementations provides unique properties due to the language features and the precise design decisions. For instance, the C runtime is particularly well-suited for bare-metal execution; the C++ target uses high-level language features and is designed to efficiently exploit parallelism by mapping triggered reactions to a pool of worker threads; and the TypeScript target is designed to integrate with the event loop of Node.js.[26]

Depending on the available language features, each target provides unique challenges and opportunities for code generation. For instance, C and Python do not provide a strong type system or other language features that allow for preventing accidental violations of reactor semantics in reaction code. For instance, reaction bodies are not allowed to access state outside their reactor, but there is no reliable way to prevent access to shared state in C or Python.

In the C++ and Typescript targets, reactors are mapped to classes, and the code generators can use various language features to properly encapsulate state, control what is accessible from a reaction body, and enforce the immutability of shared data. For these targets, our measures are sufficient for detecting most accidental violations of reactor semantics at compile time or at runtime. The Rust target can give even stronger guarantees. The strong type system and the guaranteed memory safety of Rust effectively prevent concurrent access to mutable shared state.[27] Given the strong type system of Rust, a verified runtime is also within reach.[28]

24: Lohstroh, E. A. Lee, Bateni, et al. 2023, *Lingua Franca*.

25: Bateni, E. A. Lee, et al. 2023, *reactor-c*; Menard and Tanneberger 2023, *reactor-cpp*; Fournier and Hayeß 2023, *reactor-rs*; Lohstroh, H. Kim, et al. 2023, *reactor-ts*.

26: Tilkov and Vinoski 2010, *Node.js: Using Javascript To Build High-Performance Network Programs*.

27: Fournier 2021, *A Rust Backend for Lingua Franca*.

28: Denis, Jourdan, and Marché 2022, *Creusot: A Foundry for the Deductive Verification of Rust Programs*; Hayeß 2023, *Verifying the Rust Runtime of Lingua Franca*.

4.6.3 Diagram Synthesis

The LF toolchain places a strong emphasis on *pragmatics* to enhance modeling productivity. A key enabler of pragmatics is the ability to automatically synthesize graphical views from a textual model that represents the "ground truth."[29] The LF toolchain uses *transient views*[30] and *automatic layout*[31] provided by the KIELER Lightweight Diagrams framework[32] to automatically synthesize diagrams from LF code. In fact, all the diagrams showing reactor programs in this thesis were automatically synthesized from LF code.

The diagrams are created on the fly while the programmer writes code in the IDE. This graphical representation allows for a more intuitive understanding of the topology of connected reactors, particularly when the topology is complex and there are multiple levels of nested instances. Programmers can use the diagram view to focus on different aspects of LF programs by interacting with the diagram and configuring its level of detail. In the default configuration, the LF diagrams aim at providing an abstracted view of the program, and hence details such as the actual code of a reaction are hidden.

In the diagram view of the IDE, reactor instances can be interactively expanded or collapsed to reveal or hide their nested content. This allows the programmer to drill from an abstract overview of the top-level topology of the system down into the dependency relations of individual reactions in reactor instantiations. The diagrams also provide a mechanism for navigating the LF code. When clicking on elements like ports or reactions in the diagram, the text editor jumps to the corresponding position in the LF code. The diagram synthesis also provides valuable support for debugging certain issues in LF programs, in particular causality loops. If a dependency cycle is detected in the LF program, then the involved dependencies are highlighted

29: Fuhrmann and Hanxleden 2010, *Taming Graphical Modeling*; Hanxleden, E. A. Lee, et al. 2022, *Pragmatics Twelve Years Later: A Report on Lingua Franca*.

30: C. Schneider, Spönemann, and Hanxleden 2013, *Just Model! – Putting Automatic Synthesis of Node-Link-Diagrams into Practice*.

31: Schulze, Spönemann, and Hanxleden 2014, *Drawing Layered Graphs With Port Constraints*.

32: The KIELER Project 2023a, *KIELER Lightweight Diagrams*.

in the diagram, and the interface provides additional filtering options to solely focus on the elements in a cycle.

4.6.4 IDE support

Currently, the LF ecosystem supports two IDEs. The first IDE is an Eclipse-based IDE called Epoch.[33] It heavily relies on Xtext's capability for automatically generating Eclipse plugins. Epoch is a standalone application that includes the complete LF compiler. It provides all the basic functionality one would expect from an IDE: code highlighting and completion, compilation, error feedback, and code navigation. Epoch also seamlessly integrates the diagram synthesis.

33: Lohstroh, Schulz-Rosengarten, et al. 2023, *Epoch IDE for Lingua Franca*.

Xtext also abstracts its implementation from Eclipse through the Language Server Protocol (LSP), which facilitates integrations into various popular IDEs.[34] The LF toolchain extends the LSP protocol with support for interactive diagrams to match the seamless integration of code navigation and diagram exploration offered by Epoch. We hence refer to this implementation as a Language and Diagram Server (LDS).

34: Bünder 2019, *Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages*.

The second major IDE supported by LF is Visual Studio Code. LF support can be added to Code by downloading the Lingua Franca extension from the marketplace.[35] The extension bundles the LF language and diagram server and the complete LF toolchain. The extension supports the common editing features. However, it additionally facilitates the polyglot nature of LF by embedding partial support for various target languages in the editor. Although the LF compiler does not parse target code, the language server supplements feedback from the LF compiler with code analyses from target language compilers. Since the KIELER framework also provides a Visual Studio Code extension,[36] the LF extension integrates full support for LF's diagram generation features.

35: Donovan and Lohstroh 2023, *Lingua Franca extension for Visual Studio Code*.

36: The KIELER Project 2023b, *KLighD for the Web*.

4.6.5 Command Line Tools

The LF ecosystem also provides a set of command line tools, that allow for interacting with the toolchain without the need for an IDE with a graphical user interface. Currently, there are three tools available: `lfc`, `lfd`, and `lfd`. `lfc` provides a command line interface for the LF compiler. It compiles all LF programs provided as command line arguments and reports the progress as well as any errors or warnings in the command line output. `lff` is the Lingua Franca formatter. It automatically converts given LF programs to a standardized format. Finally, `lfd` can be used to generate diagrams on the command line.

4.7 C++ Runtime and Code Generator

This section presents the C++ runtime and the code generator in more detail. The C++ runtime is designed to exploit parallel resources efficiently and assumes that it is running on top of an OS. The C runtime, by contrast, is optimized for keeping a low footprint in terms of code and memory size and can be compiled for deeply embedded devices. While the C runtime also provides multithreaded execution, it has not been fully optimized for this use case. Moreover, the C runtime requires that the complete program structure be known statically. Any reconfiguration of an LF program using the C target also requires recompilation. The C++ target, in contrast, is designed to allow

for more flexibility, so that parameter values can be changed at program invocation even when this changes the program structure (cf. Section 5.1).

4.7.1 C++ Runtime

The C++ runtime reactor-cpp is designed as a standalone library.[37] It can be used fully independently of Lingua Franca for describing reactor programs in C++. This has two advantages. First, it allows other tools and frameworks to build on top of the reactor library without the need to use Lingua Franca. Second, it creates a clear interface between the LF code generator and the concrete runtime implementation.

37: Menard and Tanneberger 2023, *reactor-cpp*.

The C++ runtime defines several classes that represent the elements of a reactor. The Unified Modeling Language (UML) diagram in Figure 4.17 gives an overview of the core classes provided by the runtime. `ReactorElement` is the universal base class from which each of the elements of a reactor derives. Most importantly, this base class assigns a name to each element and also gives each element a pointer to its containing reactor. In addition, it defines the virtual methods ***`startup`***`()` and ***`shutdown`***`()`. These methods are invoked by the runtime before the execution starts and before the execution terminates, respectively. They can be overridden by child classes that implement specific reactor elements, e.g., to perform the scheduling of initial events at startup or to schedule the shutdown trigger at termination.

A `Reactor` is a special `ReactorElement` that may contain arbitrary other components. It overrides the ***`startup`***`()` and ***`shutdown`***`()` methods such that it calls the methods recursively on all contained elements. In addition, the `Reactor` class defines the abstract method `assemble()`, which provides the main mechanism for constructing reactor programs. The `Reactor` class serves as a template for the implementation of concrete reactors that derive from the `Reactor` base class, add arbitrary reactor elements, and provide an implementation of the `assemble()` method.

The concrete implementation of `assemble()` is expected to register all elements contained by the reactor, establish any connections between ports within the reactor's scope, and declare the triggers, sources, and effects of contained reactions. The runtime invokes `assemble()` recursively in the initialization phase before the execution of the actual reactor program starts. The scheduler leverages the information gathered from the `assemble()` methods to construct the APG and assign a level to each reaction. Since the APG is constructed at runtime based on the concrete instantiated program structure, no compile-time knowledge of the entire program structure is required.

Other classes that implement `ReactorElement` are `Reaction`, `BaseAction`, and `BasePort`. Each `Reaction` has a level, a body, and optionally a deadline and a deadline handler. `BaseAction` and `BasePort` are abstract classes that represent ports and actions. They provide an abstraction over concrete actions and ports that are typed. `BasePort` provides a `bind_to` method that can be used to create a new connection. It also defines the abstract method `cleanup()`, which is overridden by `Port`. This method is used for resetting the port's present flag as well as its value before the scheduler advances to the next tag.

In the terminology of this runtime implementation, any reactor element that may produce new events is considered an action. Consequently, `Timer`, `ShutdownTrigger`, and `StartupTrigger` also implement `BaseAction`. However, only the classes `LogicalAction` and `PhysicalAction` actually expose the `schedule` method to the user. The `BaseAction` class defines the methods

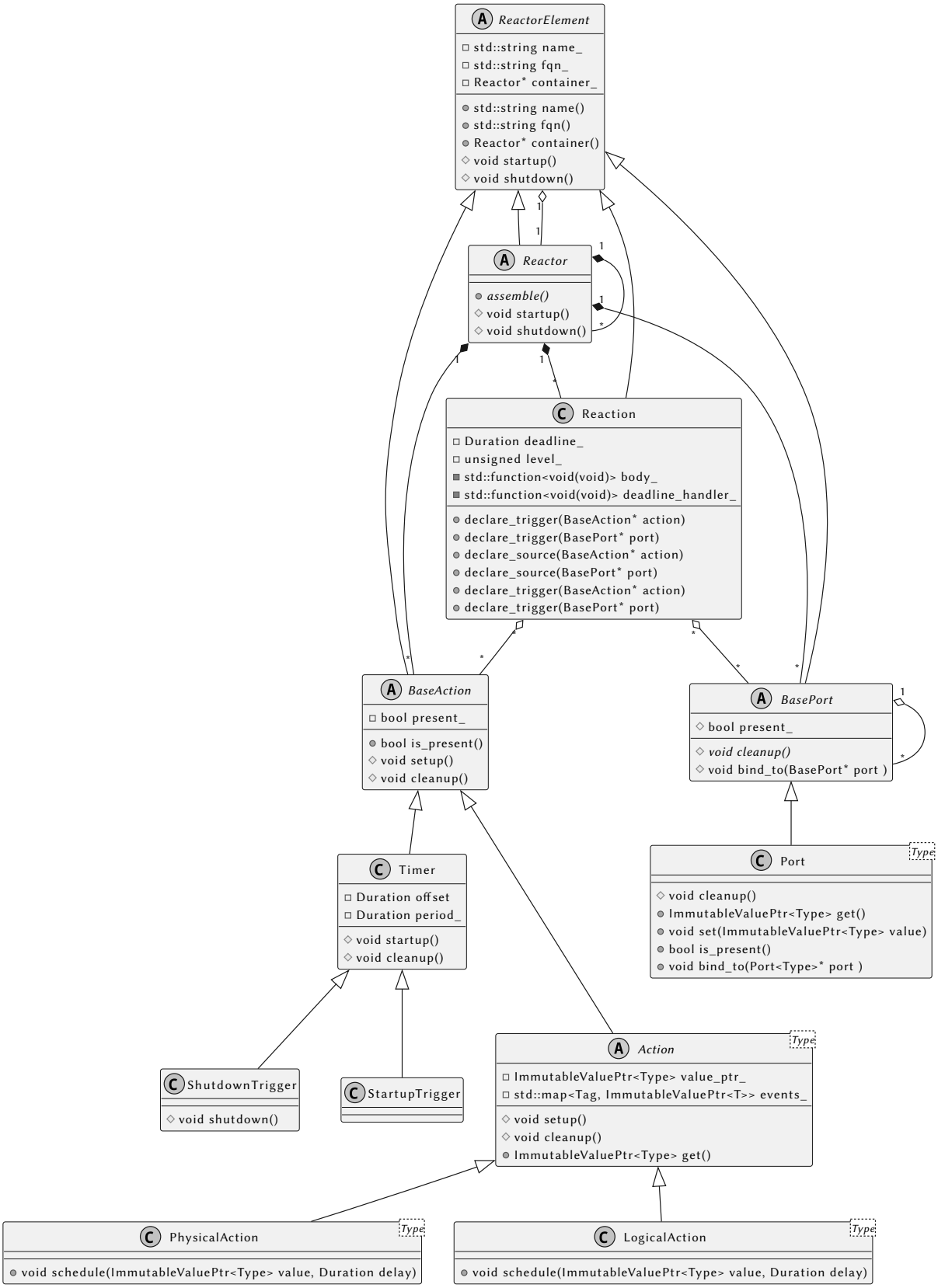


Figure 4.17: UML diagram showing the core classes of the C++ reactor runtime.

`setup()` and `cleanup()` which can be overridden by concrete implementations. `setup()` is called by the scheduler right after it advances logical time to the next tag. This may be used to initialize the action with its value if it carries an event at the tag. Similarly, `cleanup()` is called by the scheduler after all reactions at the current tag have completed their execution. It is commonly used to clear the present flag and reset the current value of the action if applicable. In addition, `Timer` uses the `cleanup()` method to schedule its next triggering.

The UML diagram does not show the classes `Environment` and `Scheduler`. While the latter provides the scheduler implementation, the former serves as the main entry point for reactor programs. The environment keeps references to the top-level reactors and also stores the APG as well as all other information that is required for managing reactor programs.

4.7.2 Ownership Types

The C++ runtime provides two class templates called `ImmutableValuePtr` and `MutableValuePtr` that implement smart pointers.[38] They are used for modeling ownership and access restrictions of values stored in ports and actions.

38: Edelson 1992, *Smart Pointers: They're Smart, But They're Not Pointers*; Dmitrović 2023, *Smart Pointers*.

`ImmutableValuePtr` is a wrapper around an `std::shared_ptr`.[39] Similarly to `std::shared_ptr` it allows for shared access to an object, and multiple references may exist. However, `ImmutableValuePtr` only provides **`const`** access to the underlying object and hence enforces immutability. This is important for avoiding unexpected data races when multiple reactors receive the same value. All values returned when calling `get()` on a port or action are wrapped in an `ImmutableValuePtr`

39: cppreference.com 2023b, *std::shared_ptr*.

`MutableValuePtr` is a wrapper around an `std::unique_ptr`.[40] It ensures that there is only a single reference to the object, but the owner of the object may mutate it. This is convenient for creating new values and modifying them, before they are sent to the next reactor via a port. When passing a `MutableValuePtr` to the `schedule` method of an action or the `set` method of a port, the `MutableValuePtr` needs to be moved, which implicitly creates a new `ImmutableValuePtr` and invalidates the old reference. If a reaction needs to modify the data that it receives, it can also create a `MutableValuePtr` from an `ImmutableValuePtr`. This, however, requires copying the underlying data in case there is more than one reference.

40: cppreference.com 2023c, *std::unique_ptr*.

4.7.3 Code Generator

The LF code generator for the C++ target effectively serializes an LF program. Each construct in the LF language is converted to its equivalent concept in the runtime implementation. In particular, the code generator produces a C++ header and implementation file for each reactor definition. It defines a class that inherits from `Reactor` and represents the concrete reactor as defined in LF code. The precise contents of the `assemble` method are automatically synthesized based on the reactor definition. In particular, this includes the creation of connections and the declaration of reaction dependencies.

In addition to the code for each reactor, the code generator synthesizes a main function as the main entry point of the program. The main function is responsible for parsing all command-line arguments. For instance, properties like the fast flag, the number of worker threads, or a timeout may be specified as command-line arguments. In addition, the C++ the code generator injects

code for parsing command-line arguments that represent the parameters of the main reactor. After parsing the command-line arguments, the main function creates a new reactor environment and instantiates the main reactor. Finally, it assembles the program and initiates the execution. Listing 4.18 shows a simplified main function that could be generated for a fictive Hello World program.

```cpp
#include "reactor-cpp/reactor-cpp.hh"

// Include the HelloWorld reactor.
#include "HelloWorld/HelloWorld.hh"

int main(int argc, char **argv) {
  bool fast{false};
  reactor::Duration timeout = reactor::Duration::max();
  unsigned workers = std::thread::hardware_concurrency();

  /* Parse command line arguments ... */

  // Create a new environment.
  reactor::Environment e{workers, fast, timeout};

  // Instantiate the main reactor.
  auto main = std::make_unique<HelloWorld>(
    "HelloWorld", &e, /* parameters */);

  // Assemble the reactor program.
  e.assemble();

  // Start execution.
  auto thread = e.startup();

  // Wait until the execution completes.
  thread.join();

  return 0;
}
```

Listing 4.18: A simplified main function, as it could be generated for an LF "Hello, World!" program.

4.8 Conclusion

This chapter gave a broad introduction to Lingua Franca, a polyglot language that enables deterministic coordination across multiple timelines. The syntax and semantics of LF closely resemble the reactor model. In contrast to a native implementation in the target language, e.g., using the DEAR framework directly in C++, LF makes the definition of reactor programs more accessible. The LF compiler can also statically check various properties of reactor programs and implement stricter scoping rules that prevent accidental violations of the reactor semantics. In addition, LF enables a separation of concerns, where the structure and coordination of components are modeled in LF and the business logic is implemented in the target language.

The discussions in this chapter include several code examples that illustrate how LF can be used to solve various problems. In particular, the presented discussions focus on LF's ability to coordinate the execution of programs deterministically. LF considers both a logical and a physical timeline, and also coordinates federated programs across multiple timelines. In contrast to many existing languages and frameworks, LF provides deterministic semantics by default but also allows for the introduction of nondeterminism, similar to the Hewitt actor model, where needed.

This chapter illustrated the reactivity, timed semantics and determinism of LF. These properties, combined with the comprehensive tooling that is available,

make Lingua Franca particularly suitable for designing CPS programs. The following chapters consider the scalability of LF programs in more detail.

Efficient Deterministic Concurrency

5

The actor model is widely accepted for programming large concurrent applications, and implementations such as the C++ Actor Framework (CAF)[1] and Akka[2] are known to be fast and efficient at utilizing a larger number of parallel cores. Compared to actors, Lingua Franca imposes various restrictions that amount to a MoC in which fewer behaviors are allowed. This chapter demonstrates that the deterministic ordering of events do not necessarily introduce overhead or higher execution times. In fact, LF is considerably faster than Akka or CAF for many benchmarks.

The core of this chapter is an extensive performance evaluation based on the Savina[3] actor benchmark suite. For this evaluation, we ported most of the Savina benchmarks to LF. Many of the Savina benchmarks are significantly larger than any of the example programs discussed in earlier chapters. To provide mechanisms for expressing such programs conveniently in LF, Section 5.1 introduces a syntax extension for LF that allows expressing scalable connection patterns. Section 5.2 provides insights on various optimizations that were implemented in the C++ reactor runtime to avoid unnecessary bottlenecks and achieve a performance that is comparable to and even exceeds that of actor frameworks. Finally, Section 5.3 presents and discusses measurement results that were conducted using the Savina benchmark suite.

The language extension, optimizations, and results presented in this chapter were published before in Menard, Lohstroh, et al. 2023, *High-Performance Deterministic Concurrency Using Lingua Franca*.

1: Charousset, Hiesgen, and T. C. Schmidt 2016, *Revisiting Actor Programming in C++*.

2: Roestenburg, Williams, and Bakker 2016, *Akka in Action*.

3: Imam and Sarkar 2014, *Savina – An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries*.

5.1 Scalable Connection Patterns in LF

The syntax of Lingua Franca, as introduced in Section 4.2, requires that all reactor instances, ports and connections are listed individually. This may become tedious for larger programs. Consider again, for example, the account program in Listing 4.3 on Page 65. The `Account` reactor defines two individual input ports, one for each user, and the main reactor instantiates and connects the `User` reactors individually.

To scale the account example to four users, we would need to add two additional input ports to the `Account` reactor and two additional instantiations and connections to the main reactor. This explicit listing of all ports, connections and instances is not only cumbersome for the programmer; it also means that the LF code needs to be adjusted and recompiled whenever the problem size changes.

To address this problem, this section introduces a syntax extension for creating multiple ports or reactor instances at once. Further, this section introduces an overloading of LF's connection operator to create multiple connections at once. This mechanism allows realizing various complex connection patterns in a single line of code and in a parameterizable way, such that LF programs can transparently scale to a given problem size without recompilation. This is a key enabler for implementing the programs of the Savina benchmark suite, which we use in Section 5.3 to evaluate LF's performance.

5.1.1 Banks and Multiports

Concretely, the extended syntax adds an optional width specification in brackets to the **input**, **output**, and the **new** keyword. If a width is specified, this syntax creates an array of ports or an array of reactor instances. We call such an array of ports a *multiport* and an array of reactor instances a *bank*. We further extend the connection operator, such that multiple ports may be listed on either side of the operator in a comma-separated list. Finally, the syntax extension introduces the *broadcast* modifier `(...)+` and the `interleaved` modifier, which provide more control over how the listed ports are connected. Listing 5.1 summarizes the modified syntax rules.

```
1  width: '[' expression ']';
2
3  port: ('input' | 'output') width? name=ID (':' type)?;
4
5  instantiation:
6    name=ID '=' 'new' width? reactorClass=ID
7    '(' (assignment (',' assignment)*)? ')';
8
9  connection:
10   (port_references | '(' port_references ')+')
11   ('->' | '~>')
12   port_references ('after' delay=expression)?;
13
14 port_references: port_reference (',' port_reference)*;
15 port_reference: reference | 'interleaved(' reference ')';
```

Listing 5.1: An extension to the Lingua Franca syntax given in Listing 4.2, providing support for banks and multiports.

Using this syntax, we can derive a scalable implementation of the simple account example in Listing 4.3. The code and diagram of the modified version are shown in Listing 5.2. Instead of defining individual inputs, the `Account` reactor defines a multiport input called `request` on line 5. The width of this multiport is given by the `num_users` parameter.

The reaction on line 7 triggers if any of the individual ports in the multiport carry an event. If multiple ports carry an event at the same tag, then the reaction is only triggered and executed once at this tag. Since we cannot know in advance which ports actually carry an event at a particular tag, the reaction body iterates over all ports, checks if a value is present, and then calls the `apply` method for each present request. This small modification allows the `Account` reactor to interact with an arbitrary number of users and truly separates the business logic as implemented in the reaction from how `Account` is used in the system.

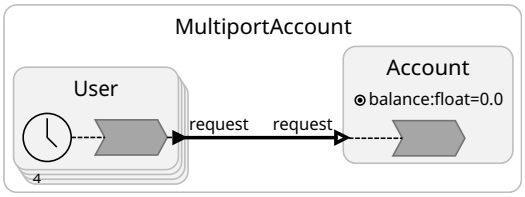
Instead of creating individual users, the main reactor instantiates a bank of `User` reactors on line 38. The main reactor also defines the parameter `num_users`, which it passes to the account instance and uses to specify the width of the user bank. On line 27, the `User` reactor defines a parameter called `bank_index` When a reactor is instantiated in a bank and defines a `bank_index` parameter, then the runtime automatically assigns the index of each instance within the bank to the parameter. This allows for the state or the behavior of a reactor to depend on its position within a bank. In this example, each user sends a request at startup with a value that is calculated using the `bank_index` parameter.

The connection operator on line 40 connects the request outputs of all the users to the multiport input of the account. Thereby, it connects the output of the *n*th user to the *n*th port in the input multiport of the account. This pattern implements many-to-one communication.

If the number of ports on the left-hand side and the right-hand side of the connection operator do not match, then some ports remain unconnected.

Listing 5.2: A scalable implementation of the simple account example given in Listing 4.3 using banks and multiports.

```
target Cpp

reactor Account(num_users:size_t = 4) {
  state balance: float = 0.0
  input[num_users] request:float

  reaction(request) {=
    for (size_t i{0}; i < num_users; i++) {
      if (request[i].is_present()) {
        apply(i, *request[i].get());
      }
    }
  =}

  method apply(user_id:size_t, value:float) {=
    std::cout <<"Request for" << value
              << " from user" << user_id <<" was ";
    if (balance + value >= 0) {
      balance += value;
      std::cout <<"accepted.\n";
    } else {
      std::cout <<"denied.\n";
    }
  =}
}

reactor User(offset:time = 0,
             bank_index:size_t = 4) {
  timer t(offset)
  output request: float

  reaction(t) -> request {=
    request.set(15.0 - bank_index * 10.0);
  =}
}

main reactor(num_users:size_t=4) {
  account = new Account(num_users=num_users)
  users = new[num_users] User()

  users.request -> account.request
}
```



Let n and m denote the number of ports on the left and right, respectively. Only the first $\min(n, m)$ ports are connected on either side. If a port remains unconnected, a warning is issued.[4]

4: If the widths are known statically, the LF validator issues a warning at compile-time. Otherwise, the runtime issues a warning during the initialization phase.

When executing the example program in Listing 5.2, it prints:

```
Request for 15 from user 0 was accepted.
Request for 5 from user 1 was accepted.
Request for -5 from user 2 was accepted.
Request for -15 from user 3 was accepted.
```

Note that the number of users can be adjusted arbitrarily. Since `num_users` is a parameter to the main reactor, the LF code generator will also add it to the program's command line arguments, which allows overwriting the default parameter without recompilation. When running, for example, the command `./bin/MultiportAccount --num_users 6`, the program prints:

```
Request for 15 from user 0 was accepted.
Request for 5 from user 1 was accepted.
Request for -5 from user 2 was accepted.
Request for -15 from user 3 was accepted.
Request for -25 from user 4 was denied.
Request for -35 from user 5 was denied.
```

5.1.2 Connection Patterns

The syntax extension for supporting banks and multiports in LF programs is relatively simple. Yet, it is powerful enough to cover many communication patterns. The following discusses a selection of common patterns that can be conveniently expressed in LF.; The presented patterns are extensively used in the Savina benchmark implementations.

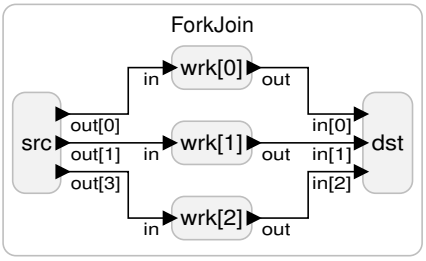
Note that the following example programs do not include reactions or other implementation details and solely focus on the connection patterns. The LF diagram synthesis does not show individual reactor or port instances. Therefore, the below diagrams were manually "desugared" to visualize the underlying patterns.

Fork-Join

Listing 5.3 gives an example program implementing a fork-join pattern, which combines one-to-many and many-to-one communication. The program defines a `Source`, a `Worker`, and a `Sink` reactor. `Source` defines a multiport output of width `n`, and `Sink` defines a multiport input of width `n`. `Worker`, however, only defines a single input and output port. The `Worker` reactor is instantiated in a bank of width `n` by the main reactor. The two connection statements in the main reactor each establish `n` connections, one for each pair of multiport and bank instances. The desugared diagram on the right visualizes the individual connections and instances.

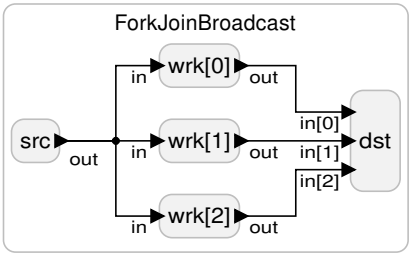
Listing 5.3: A fork-join pattern using one-to-many and many-to-one connections in LF.

```
1  target Cpp
2  reactor Source(n: size_t = 3) {
3    output[n] out: int
4  }
5  reactor Worker {
6      input in: int
7      output out: int
8  }
9  reactor Sink(n: size_t = 3) {
10   input[n] in: int
11 }
12 main reactor(n: size_t = 3) {
13   src = new Source(n=n)
14   dst = new Sink(n=n)
15   wrk = new[n] Worker()
16
17   src.out -> wrk.in
18   wrk.out -> dst.in
19 }
```



In this example, the source reactor has to produce three separate values and send them to the workers individually. The broadcast modifier `(...)+` can be used instead to broadcast a single value to all workers. The program in Listing 5.4 configures the source reactor to use only a single output by setting `n=1`. By using the broadcast modifier on line 11, we can replicate `src.out` and connect it to the inputs of all workers.

```
1  target Cpp
2
3  import Source, Sink, Worker
4  from "ForkJoin.lf"
5
6  main reactor(n: size_t = 3) {
7    src = new Source(n=1)
8    dst = new Sink(n=n)
9    wrk = new[n] Worker()
10
11   (src.out)+ -> wrk.in
12   wrk.out -> dst.in
13 }
```



Listing 5.4: A fork-join pattern using a broadcast connection in LF.

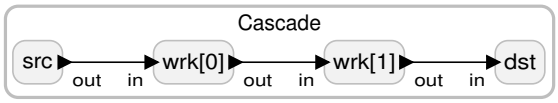
In either variant, any reactions contained in one of the workers may execute in parallel to the reactions of all other workers.

Cascade Composition

Using banks and multiports, we can also conveniently compose cascades of reactors. This is illustrated by the program in Listing 5.5. The connection operator sequences all ports listed on the left- and right-hand sides and connects the *n*th port on the left-hand side to the *n*th port on the right-hand side. By offsetting the left-hand side of the connection statement in line 10 with a single source port and appending the sink port to the right-hand side, we can effectively arrange the connections to form the cascade shown in the diagram on the right.

Listing 5.5: A cascade pattern using a bank in LF.

```
1  target Cpp
2
3  import Source, Sink, Worker from "ForkJoin.lf"
4
5  main reactor(n: size_t = 2) {
6    src = new Source(n=1)
7    dst = new Sink(n=1)
8    wrk = new[n] Worker()
9
10   src.out, wrk.out -> wrk.in, dst.in
11 }
```



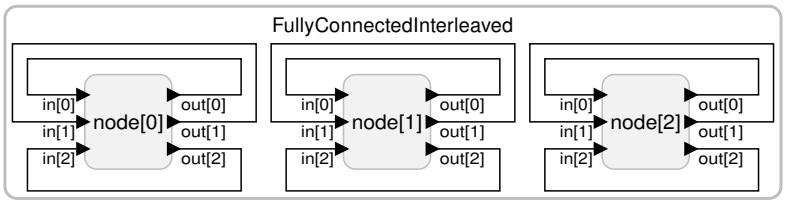
Fully Connected

The connection operator also connects multiports within banks. In this case, the operator will implicitly unfold all port instances on both sides of the connection to form a flat list of ports. The unfolding happens such that we first list all ports of the first bank instance, then all ports of the second instance, and so on. In principle, we can use this to connect all reactor instances within a bank to all other instances to create a fully connected graph.

The program in Listing 5.6 attempts to create such a pattern. However, the connection operator on line 10 uses the same unfolding strategy on either side of the connection. When we consider the diagram, it becomes obvious that the resulting pattern is not very useful. Each reactor instance is connected to itself.

Listing 5.6: An attempt at many-to-many communication in LF.

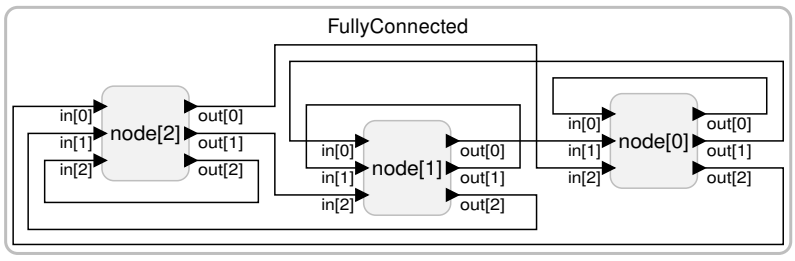
```
1  target Cpp
2
3  reactor Node(n: size_t = 3) {
4    input[n] in: int
5    output[n] out: int
6  }
7
8  main reactor(n: size_t = 3) {
9    node = new[n] Node()
10   node.out -> node.in
11 }
```



Using the `interleaved` modifier on either side of the connection, we can modify the unfolding strategy. The interleaved strategy first lists all first port instances within all bank instances, then the second port instances within all bank instances, and so on. By applying `interleaved` to the ports on the right side of the connection operator, the program in Listing 5.7 creates the desired fully connected pattern. This allows each node to send and receive messages to and from all other nodes. Thereby, the *n*th output or input port corresponds to the *n*th instance of the node.

Listing 5.7: Many-to-many communication in LF using the `interleaved` modifier.

```
1  target Cpp
2
3  import Node from "FullyConnected.lf"
4
5  main reactor(n: size_t = 3) {
6    node = new[n] Node()
7    node.out -> interleaved(node.in)
8  }
```



5.2 Optimized Reactor Scheduler

Section 3.1.4 discusses the principle strategy for executing reactor programs and provides a pseudo code implementation of the main scheduling procedure in Listing 3.3. While we can use this as a blueprint for an actual scheduler implementation, the resulting scheduler will likely be inefficient. This section discusses various optimizations that were implemented in the C++ reactor runtime in order to exploit parallelism more efficiently and avoid performance bottlenecks.

5.2.1 Sorting the APG

As discussed in Section 3.1.3, we can generate an APG for each reactor program to capture the dependency constraints between reactions. Figure 5.1, for instance, shows the dependency graph for the account example program with a proxy delay in Listing 4.5. The scheduler can use this graph to identify which reactions may execute in parallel and when to wait for previous reactions to complete. The algorithm in Listing 3.3 assumes the existence of a procedure called GETREADYREACTIONS that filters the set of triggered reactions and returns a set of ready reactions. Implementing this procedure efficiently, however, is challenging.

In order to identify if a reaction is ready for execution, we need to traverse the graph and check if all dependent reactions are either not triggered or have completed executing. This operation can be costly if the APG is large, especially since the scheduler needs to perform it repeatedly for each reaction. The C++ runtime, instead, implements a conservative approach. In the initialization phase, the APG is sorted by *level* (also called top level).[5] The level of a reaction is the length of the longest path to the reaction from any root of the graph. The graph in Figure 5.1 is sorted by level.

5: Kwok and Ahmad 1999, *Static Scheduling Algorithms for Allocating Directed Task Graphs To Multiprocessors*.

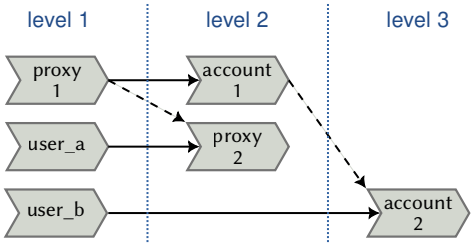


Figure 5.1: The APG for the account example program with a proxy delay in Listing 4.5.

Using the assigned levels, we know that any two reactions with the same level may be executed in parallel. However, if reactions have different levels, then we must assume that there is a dependency from the reaction with the higher level to the reaction with the lower level. Therefore, the strategy is conservative. The level assignment allows deciding quickly if two reactions may execute in parallel, but it may miss opportunities for exploiting parallelism. For instance, the second reaction of account in principle could execute in parallel to the second proxy reaction, but when only considering the level, we must assume that there is a dependency.

Using the level mechanism, implementing the GETREADYREACTIONS procedure becomes trivial. Listing 5.8 shows one possible implementation. It assumes a global variable that stores the current level. Furthermore, it assumes that GETREADYREACTIONS is only called once all reactions from the previous level have finished executing. If the reaction queue Q_R is implemented as a map of levels to a list of reactions, then filtering for all reactions with a specific level becomes trivial and very efficient.[6]

6: We can also use an array where the index represents the level.

```
1: procedure getReadyReactions(Q_R)
2:     R ← ∅
3:     while level < maxLevel ∧ R = ∅ do
4:         for all r ∈ Q_R do
5:             if getLevel(r) = level then
6:                 R ← R ∪ r
7:         level ← level + 1
8:     return R
```

Listing 5.8: Level-based implementation of the getReadyReactions procedure.

The level-based scheduling approach was described before by Lohstroh[7] and is used in all the runtime implementations that are available for Lingua Franca. While the strategy is conservative, it is sufficient to exploit parallelism in most cases, as the evaluation in the next section shows.

7: Lohstroh 2020, *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*.

5.2.2 Coordinating Worker Threads

Figure 3.4 and the discussion in Section 3.1.4 conceptually distinguish the scheduler and the worker threads. In an actual implementation, however, using a central scheduler and separate worker threads introduces several synchronization points. The scheduler needs to send work to the workers, and the workers need to notify the scheduler when they are finished. Instead, in our implementation, a thread that runs out of work tries to become the scheduler and moves ready reactions to the ready queue or advances logical time to the next tag if all reactions have been processed. Only one worker thread can become the scheduler, and all other workers that run out of work will go to sleep until they are woken up again by the scheduler.

```
1: procedure work()
2:     if workerID = 0 then
3:         ▷ The worker with ID 0 performs the initial scheduling. ◁
4:         runScheduler()
5:     while True do
6:         ▷ Retrieve a reaction from the ready queue. This may block until
           a reaction becomes available. ◁
7:         r ← readyQueuePop()
8:         execute(r)                          ▷ Execute the reaction.
9:         ▷ Decrement the atomic counter n_R, which represents the number
           of remaining ready reactions to be processed. ◁
10:        if atomicDecrement(n_R) = 1 then
11:            ▷ If the atomic decrement returns a previous value of 1, then
               it processed the last ready reaction. ◁
12:            runScheduler()      ▷ Schedule the next round of reactions.
```

Listing 5.9: Main work function executed by each worker.

Listing 5.9 shows the work procedure, which is the main function that each worker thread executes.[8] Determined by the thread ID, one worker calls the runScheduler procedure to perform the initial scheduling (line 4). Each worker enters an infinite while loop. In this loop, the worker thread first attempts to retrieve a reaction from the ready queue (line 7). This operation may block if there are no more reactions in the queue. Thus, the worker thread will wait until it can retrieve a ready reaction from the queue. Once the worker thread has retrieved a ready reaction, it executes the reaction.

8: The shown procedure is simplified and does not include details like termination.

When the reaction execution completes, the worker thread decrements the atomic variable n_R, which denotes the number of ready reactions that have not yet finished executing (line 7). The atomicDecrement procedure returns the previous value of the variable (before decrementing). Thus, if the atomic

decrement returns 1, we know that the current worker thread just completed executing the last ready reaction. It then calls runScheduler to schedule the next round of reactions. Since this point can only be reached by one worker thread and the other worker threads can only continue execution once we fill the ready queue, we have achieved mutual exclusion, and there is no need for using locks.

5.2.3 Lock-free Data Structures and Algorithms

The reaction and event queues and other data structures that are required for bookkeeping (e.g., a list of all set ports) are shared across worker threads. To avoid data races, access to these data structures needs to be guarded. However, using mutexes and locks for synchronization proved to be inefficient due to high contention on the shared resources, especially when many parallel reactions set ports or schedule actions. Instead, the C++ scheduler utilizes lock-free data structures and algorithms where possible. One example of this was already given by the work procedure Listing 5.9, which uses a non-blocking atomic decrement operation to determine when to schedule and to guarantee mutual exclusion between the scheduling thread and the other worker threads.

The ready queue is implemented as a fixed-size array paired with an atomic counter $n_{\mathscr{R}}$ that denotes the number of ready reactions. Since we know precisely how many reactions can run at most in parallel, i.e., the maximum number of reactions in the APG that have the same level, we can fix the size of the queue. Listing 5.10 gives a pseudocode implementation of the readyQueuePop that returns a single reaction from the queue. The procedure first atomically decrements the queue size counter $n_{\mathscr{R}}$. If the old size n (from before the decrement operation) is positive, then there are remaining elements in the queue and we can return the $n-1$th element (line 8). Otherwise, the procedure enters the while loop, where it tries to acquire the counting semaphore s. Once the worker thread has acquired the semaphore, it tries again to decrement the queue size.

```
1: procedure readyQueuePop()
2:     ▷ Atomically decrement the queue size. n denotes the size before the
         decrement operation. ◁
3:     n ← atomicDecrement(n_R)
4:     while n ≤ 0 do
5:         aquireSemaphore(s)
6:         n ← atomicDecrement(n_R)
7:     ▷ Return the n − 1th element in the queue. ◁
8:     return R[n − 1]
```

Listing 5.10: Implementation of the readyQueuePop procedure.

The purpose of the semaphore is twofold. First, it provides a simple mechanism for blocking and unblocking a worker thread. Second, the counting semaphore allows for precise control of the total number of worker threads that may execute at a given point. When filling the queue, we can wake up precisely as many worker threads as there are reactions to be processed in parallel. This allows for precise control of the parallelism and avoids waking up threads unnecessarily if there is not a sufficient number of parallel reactions to utilize all worker threads.

Listing 5.11 gives an implementation of the readyQueueFill procedure, which is the counterpart to the readyQueuePop procedure. It is used by the scheduler to move the list of ready reactions at the next level obtained via GetReadyReactions to the ready queue $\mathscr{R}$.

```
1:  procedure readyQueueFill(R)
2:      R_ℛ ← R                    ▷ Overwrite the queue with the new ready reactions.
3:      ▷ Atomically store the new queue size and retrieve the old value. ◁
4:      n ← atomicExchange(n_ℛ, |R|)
5:      ▷ Update the number of waiting worker threads. ◁
6:      waitingWorkers ← waitingWorkers + |n|
7:      runningWorkers ← numWorkers − waitingWorkers
8:      ▷ Determine how many workers to wake up. ◁
9:      wakeUp ← min(waitingWorkers, |R| − runningWorkers)
10:     if wakeUp > 0 then
11:         releaseSemaphore(s, wakeUp)
```

Listing 5.11: Implementation of the readyQueueFill procedure.

The procedure first updates the ready queue. It then uses an atomic exchange operation to assign the new size of the reaction queue to $n_{\mathscr{R}}$ and simultaneously retrieve its old size and store it in the variable n. Note that the value of n could be negative. If more workers attempt to pop a reaction from the queue than there are elements in the queue, then the atomic decrement operations in readyQueuePop result in a negative value. $|n|$ denotes precisely the number of workers that attempted to pop a reaction from the queue but had to wait instead.

The scheduler holds a variable `waitingWorkers` and uses it to keep track of the number of workers that are currently in a waiting state (i.e., that are blocking on the semaphore or about to call the acquire procedure on the semaphore). Line 6 increments the number of waiting workers by $|n|$. Based on this, we can also calculate the number of workers that are still running. These are worker threads that executed past line 10 in Listing 5.9, but that did not yet attempt to retrieve the next reaction. Finally, line 9 calculates how many worker threads should be woken up. At most, this should be the number of waiting threads. Also, we should not wake up more workers than there are reactions to execute. The number of additional workers needed to execute all reactions in parallel is $|R| - $ `runningWorkers`. Therefore, the procedure takes the minimum of both and then releases the semaphore, incrementing its counter by this value.

There are many occasions where the C++ runtime utilizes lock-free data structures and algorithms to orchestrate access to shared resources. However, there are too many to explain them all in detail. The presented procedures are therefor exemplary for a range of implemented optimizations.

5.2.4 Sparse Multiports

Often, reactions that are triggered by a multiport input need to identify which ports actually have a present value. This is typically done by iterating over all ports, as shown in the example in Listing 5.2. Let n denote the width of the multiport and p the number of present ports. If the multiport width is large and communication is sparse ($p \ll n$), then iterating over all ports and checking for presence individually is inefficient ($\mathcal{O}(n)$).

The optimized C++ runtime provides additional methods on multiports that return only the present ports. Internally, the runtime uses a lock-free buffer for each multiport to keep track of the ports that are actually set at a given tag. The method `present_indices_unsorted` returns this buffer as an unsorted list of all the present indices. When using this method, iterating over all present ports in the reaction body has complexity $\mathcal{O}(p)$. However, the port indices returned by this method may have an arbitrary order if the ports are set by concurrent reactions. If a fixed order is required,

`present_indices_sorted` can be used to obtain a sorted list of indices. The sorting has complexity $\mathcal{O}(p \cdot \log(p))$.

5.3 Performance Evaluation

Using the runtime optimizations as well as the bank and multiport syntax extension discussed in the previous section, this section evaluates the performance of LF programs in the C++ target. This is done by porting a range of actor benchmarks to LF and comparing their performance to implementations using Akka and CAF.

5.3.1 Methodology

The evaluation is based on the Savina benchmark suite for actor languages and frameworks.[9] While this suite has several issues, as Blessing et al. discuss in more detail,[10] Savina covers a wide range of patterns and, to the best of our knowledge, is the most comprehensive benchmark suite for actor frameworks that has been published. The Savina suite includes Akka implementations and CAF implementations of most benchmarks are also available.

9: Imam and Sarkar 2014, *Savina – An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries*.

10: Blessing et al. 2019, *Run, Actor, Run: Towards Cross-Actor Language Benchmarking*.

22 out of the 30 Savina benchmarks were ported to the C++ target of LF. Due to the fundamental differences between the actor and reactor models, the process of porting benchmarks is not always straightforward. The porting process aimed at closely resembling the original workloads and considered the intention behind the individual benchmarks. The next subsection discusses the implementation of selected benchmarks in more detail.

The benchmarks Fork Join (actor creation), Fibonacci, Quicksort, Bitonic Sort, Sieve of Eratosthenes, Unbalanced Cobwebbed Tree, Online Facility Location, and Successive Over-Relaxation were not implemented in LF as they require the capability to dynamically create actors. In the reactor model, this can be achieved with mutations that may modify the reactor topology.[11] However, mutations are not yet fully implemented in LF, and a discussion of language-level constructs for supporting mutations is beyond the scope of this thesis. Although the precise cost of performing mutations is currently unknown, this cost will mostly depend on how efficiently the APG can be modified. Since the APG remains static in between mutations, we expect no difference in performance for the execution of reactions, and hence the results discussed here yield measurements that will be useful even when mutations are eventually supported.

11: Lohstroh, Romeo, et al. 2019, *Reactors: A Deterministic Model for Composable Reactive Systems*; Lohstroh 2020, *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*.

The presented evaluation also excludes the A*-Search and Logistic Map Series benchmarks. The A*-Search implementation in the original Savina suite suffers from a severe race condition that results in wildly varying execution times.[10] Logistic Map Series is omitted, as the Akka implementation violates actor semantics and requires explicit synchronization.[10] For this reason, the CAF implementation needs to use a blocking call, which makes it slower than the other implementations by at least two orders of magnitude. Since this is not a problem of CAF but rather a problem in the benchmark design, we omit Logistic Map Series to avoid skewing the analysis.

All measurements were performed on a workstation with an Intel Core i9-10900K processor (10 cores, 20 hardware threads) and 32 GiB DDR4-2933 RAM running Ubuntu 22.04 and using CAF version 17.6 and Akka version 2.6.17. Following the methodology of Savina, measurements exclude initialization and cleanup. Each measurement comprises 32 iterations. The first two iterations are excluded from the analysis and used for warm up.

Table 5.1: Characteristics of the Savina benchmarks implemented in LF. The middle part denotes static information about the size of the program, and the right part denotes runtime information about the execution of the program.

ID	cat.	benchmark	reactors	reactions	actions	ports	connections	processed tags	processed reactions	set ports	scheduled actions	time per reaction [ns]
1	micro	Ping Pong	4	8	3	8	4	1,000,004	3,000,005	2,000,002	1,000,010	74
2		Thread Ring	103	109	3	406	203	1,004	101,008	100,004	1,208	71
3		Counting Actor	4	11	3	12	6	1,000,005	2,000,010	1,000,005	1,000,011	77
4		Fork Join (throughput)	63	67	3	65	62	10,004	610,006	10,002	10,128	26
7		Chameneos	103	208	3	20,404	10,202	4,005	408,209	800,402	4,209	169
8		Big	123	487	122	57,964	29,042	20,004	6,472,034	4,800,122	2,400,248	221
9	concurrency	Concurrent Dictionary	24	30	3	146	83	10,004	220,028	400,023	10,050	217
10		Concurrent Sorted Linked List	24	31	4	145	83	8,005	176,029	320,022	8,051	43,323
12		Dining Philosophers	23	71	3	224	122	20,004	460,019	1,000,002	20,048	123
13		Sleeping Barber	2,005	8,017	5	24,014	12,008	4,004	18,008	14,002	8,012	1,265
14		Cigarette Smokers	203	208	4	404	202	1,005	2,007	1,002	1,409	2,692
16		Bank Transaction	1,003	3,007	3	2,004,004	1,002,002	79	78,905	101,002	2,083	464
11	parallelism	Producer Consumer (bounded)	83	209	42	484	282	1,005	122,128	160,082	40,208	12,325
17		All-Pairs Shortest Path	45	115	38	873	798	304	21,643	10,839	10,892	45,270
19		N Queens First K Solutions	23	30	4	124	62	256	5,472	9,130	300	46,150
20		Recursive Matrix Multiplication	23	50	4	105	62	37	651	661	81	1,268,569
22		Radix Sort	64	248	63	186	123	899,962	7,000,100	6,100,002	900,104	86
23		Filter Bank	54	141	3	254	150	34,821	1,073,278	809,063	34,927	2,075
28		Trapezoidal Approximation	103	108	3	404	202	5	108	202	209	6,050,557
29		Precise Pi Computation	23	30	4	84	42	213	4,584	8,322	257	22,172

5.3.2 Benchmark Implementation in LF

Table 5.1 provides an overview of all the Savina benchmarks that were ported to LF and that are included in our discussion. The table also lists various key characteristics of the LF implementations. The middle section displays characteristics about the size of each program, such as the total number of reactors, reactions, actions, ports, and connections. The right section shows details about the benchmark execution, such as the number of tags (or events) that were processed, the number of executed reactions, and how often ports were set and actions scheduled. Finally, the average time per executed reaction gives an estimate of the size of the workload implemented in each reaction.

As indicated in Table 5.1, the Savina benchmarks are divided into three categories: *micro*, *concurrency* and *parallelism*.[12] The microbenchmarks focus on stressing various mechanisms in the runtime scheduler to expose overheads in the runtime. The concurrency benchmarks have a similar goal, but they put more focus on the concurrent operation of (re)actors and also require synchronization mechanisms to solve the particular problem. Since the micro and concurrency benchmarks are designed to stress the runtime and scheduler, the workload implemented in each (re)actor is relatively small (except for Concurrent Sorted Linked List). The benchmarks in the parallelism category are designed to test the capability to exploit parallel hardware efficiently and hence the workload implemented by each (re)actor is more significant (except for Radix Sort).

12: The original Savina suite lists Producer Consumer as a concurrency benchmark, but the author finds that it fits better into the group of parallelism benchmarks.

The interested reader may find the full LF implementation of all implemented benchmarks on GitHub.[13] The remainder of this section discusses implementation details for selected, representative benchmarks.

13: Menard, Soroush, et al. 2023, *Lingua Franca Benchmarks*.

The execution of all benchmarks in the original Savina suite is governed by an actor called `BenchmarkRunner`. It is responsible for initiating a benchmark run and measuring the time until each benchmark run completes. This enables performing measurements in repeated iterations while keeping caches (and the JVM in case of Akka) warm. The LF benchmarks adapt this mechanism and use the `BenchmarkRunner` reactor shown in Figure 5.2. The runner has two ports: `start` and `finished`. While `start` is used to initiate a benchmark run, `finished` is used to receive feedback from the actual benchmark when it completes its execution.

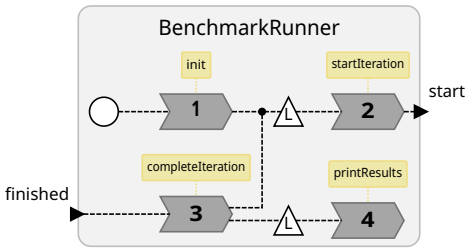


Figure 5.2: The benchmark runner implemented in LF.

The LF implementation for each of the benchmarks defines a reactor for each actor in the original Savina implementation and a connection for each message that can be sent between actors. For instance, Figure 5.3 shows the LF implementation of the Ping Pong benchmark. The benchmark consists of two (re)actors, `Ping` and `Pong`, that send each other messages back and forth. When `Ping` receives a message on the `inStart` port, it schedules a new event using its internal action. The reaction triggered by this action then sends the first ping message. `Pong` reacts to this message by sending a pong message back to `Ping`, which in turn reacts by scheduling a new event on the internal action to repeat the process. Once all 1,000,000 ping and pong

messages have been sent, the `Ping` reactor does not schedule a new event but instead notifies the benchmark runner to indicate that the benchmark execution is complete.

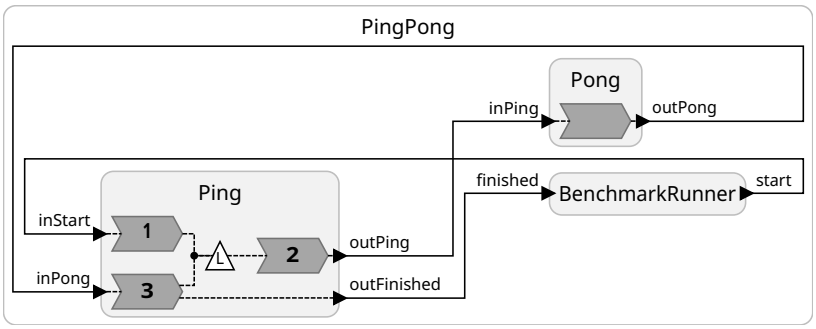


Figure 5.3: LF implementation of the Ping Pong benchmark.

Note the use of the logical action to break the dependency cycle between Ping and Pong. If we merged reactions 2 and 3 of `Ping` to send another ping message right after receiving a pong message, there would be a causality loop. The loop is broken up by scheduling a new event and sending the ping message at the next tag. All the Savina benchmarks have a direct feedback loop, and thus, where needed, logical actions were carefully inserted to break dependency cycles.

The concurrency benchmarks are particularly interesting as we can utilize LF's semantics to implement them efficiently. The Concurrent Dictionary benchmark, for instance, consists of a Dictionary (re)actor that receives read or write requests from 20 Worker (re)actors. The dictionary processes each request and sends a reply back to the workers. Figure 5.4 shows our LF implementation. It instantiates a bank of worker reactors that communicate with the dictionary via multiports. The workers operate concurrently, and each invocation of the worker reaction is logically simultaneous with the other workers. In consequence, the dictionary will receive multiple logically simultaneous requests from the workers. This notion of logical simultaneity allows the dictionary reactor to effectively batch-process all the requests received at a single tag in a single reaction. The dictionary reaction iterates over all present messages on the `request` port and processes the requests sequentially.

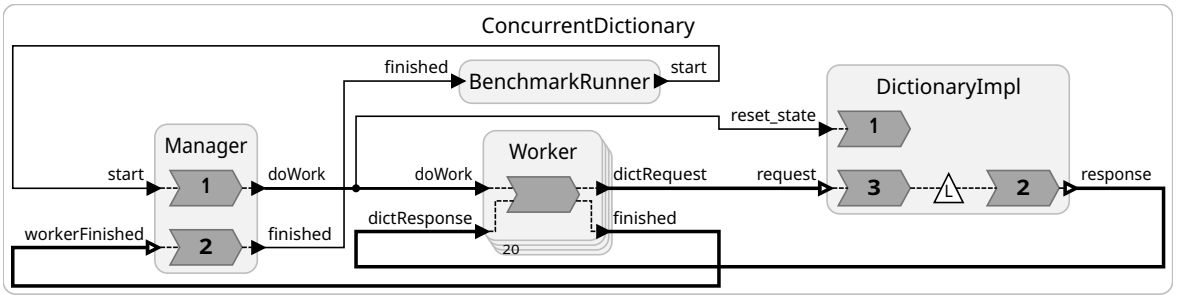


Figure 5.4: LF implementation of the Concurrent Dictionary benchmark.

In the actor implementations of the Concurrent Dictionary benchmark, however, the dictionary can only process individual requests as there is no notion of simultaneity. Thus, the runtime needs to invoke the actor behavior repeatedly, which adds additional overhead. Moreover, the particular order in which the dictionary actor processes requests s nondeterministic. Since the workers send interleaving read and write requests, they may observe different responses depending on the order in which dictionary processes the requests. LF's notion of logical time establishes a deterministic ordering

between messages and allows observing all the present inputs at a given tag at once.

We can make a similar observation for the Dining Philosophers benchmark. The LF implementation in Figure 5.5 uses an arbitrator reactor and a bank of 20 philosopher reactors. The philosophers *think* and *eat* concurrently. In order to start eating, philosophers send a `hungry`message to the arbitrator, which replies with `eat` or `denied` When a philosopher finishes eating, they indicate this with a `done`message. If the request to eat in denied, the philosopher sends a new `hungry`message.

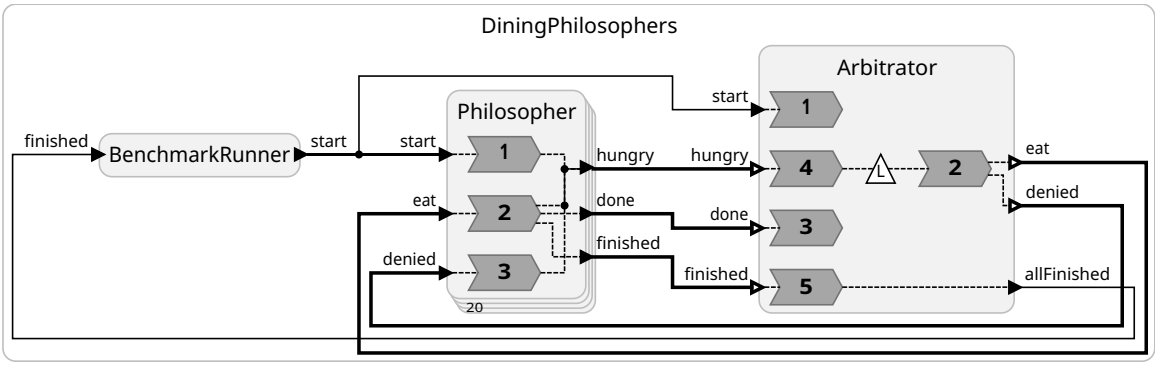


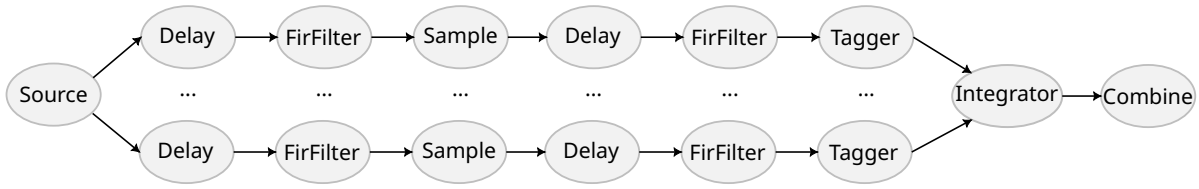
Figure 5.5: LF implementation of the Dining Philosophers benchmark.

While the philosophers operate concurrently, the arbitrator can process all logical simultaneous `hungry`requests in one batch. In this concrete benchmark, this has the additional advantage that the arbitrator always knows which philosophers are hungry at a particular tag and can therefore find a fair strategy to grant the resources to the philosophers. In an actor implementation, the arbitrator can only make decisions for individual messages, which makes it much harder to find a fair solution. The original Savina implementation "solves" this simply by having each philosopher send another `hungry` message immediately after receiving a `denied`message. This increases the chance for each philosopher to eat eventually, but it also adds a significant amount of unnecessary messages.

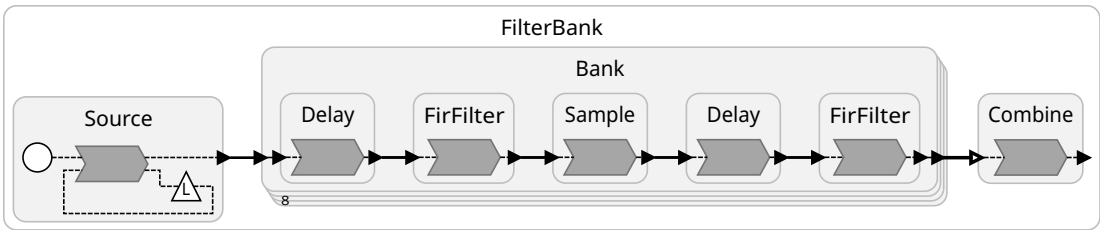
In our measurements, the Akka implementation of the philosopher benchmark used about 10 million `hungry`messages, whereas the LF implementation used about 200,000 `hungry`messages. Of course, it would be possible to implement other, more elaborate, arbitration strategies with actors, but compared to the Lingua Franca solution, this would always come with additional cost in terms of code size and also overhead for additional messages.

The LF implementation of Dining Philosophers could even be further simplified. Since there is no delay between sending an `eat` message in reaction 2 of the arbitrator, eating in reaction 2 of the philosopher, and processing the `done`message in reaction 3 of the arbitrator, all three steps are logically simultaneous. Since the runtime scheduler first completes processing all reactions at the current tag before moving to the next tag (cf. Section 3.1.4), the `done`message is redundant. When the arbitrator reaction is invoked to decide which philosopher may eat, it knows that all philosophers from the previous round must have completed eating at the previous tag. However, we decided to keep the `done`message to avoid deviating too much from the original benchmark implementation. This also allows for alternative implementations of the philosopher reactor, which might use a delay internally and send `done`messages at a later tag.

The advantage of LF's synchronous semantics also becomes evident in the Filter Bank benchmark shown in Figure 5.6. It applies a cascade of filters



(a) Actor implementation



(b) Reactor implementation in LF

Figure 5.6: Comparison between the reactor and the actor implementation of the Filter Bank benchmark.

to eight parallel channels in a data stream. The output of each filter bank is then combined into a single stream. The combine operation is applied to the *n*th output message of each bank. This is trivial in LF, as the output messages are logically synchronous. The actor implementation, however, requires an additional protocol to explicitly synchronize the outputs of the asynchronously operating banks. The original Savina implementation utilizes an additional `Taggeractor` that annotates the output messages of each bank with a tag indicating the ID of the bank. A so-called `Integratoractor` buffers the tagged messages from all banks. Once it receives a complete set of messages from all banks, it forwards them as one message to the Combine actor. As this synchronization mechanism is fully redundant in LF, it is omitted from the LF implementation of the benchmark.

5.3.3 Results and Discussion

Figure 5.7 reports measured results for all supported benchmarks obtained with Akka, CAF, and the C++ target of LF. The plots show the mean execution times (including 99% confidence intervals) for a varying number of worker threads for each of the benchmarks. Not all benchmarks are implemented in CAF, and hence CAF is missing in some plots.

The first six plots in Figure 5.7 belong to the group of microbenchmarks in the Savina suite. Overall, the optimized C++ runtime shows comparable performance to Akka and CAF. In Ping Pong and Thread Ring, the LF implementation is considerably faster than Akka but is still outperformed by CAF. For Counting Actor and Big, Akka performs better, and the LF performance is slightly behind CAF. In Fork Join and Chameneos, the LF implementation outperforms both Akka and CAF, especially when using a larger number of worker threads.

The next six plots (Concurrent Dictionary to Bank Transaction) belong to the group of concurrency benchmarks. LF significantly outperforms CAF and Akka in all the concurrency benchmarks (especially for a high number of worker threads). This highlights how concurrent behavior is expressed naturally in LF and can be executed efficiently. As discussed in the previous subsection, we can exploit the well-defined notion of logical simultaneity in LF to execute independent reactions in parallel and batch-process simultaneous messages from multiple reactors in a single reaction. Moreover, no

Figure 5.7: Mean execution times and 99% confidence intervals for various Savina benchmarks implemented in LF, CAF, and Akka, measured for a varying number of worker threads. The numbers prefixed with # are benchmark IDs as listed in Imam and Sarkar 2014, *Savina – An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries*.

explicit synchronization is needed. In the actor benchmarks, explicit synchronization, e.g., by sending acknowledge messages or using blocking calls, adds additional overhead.

The remaining plots belong to the group of parallelism benchmarks in the Savina suite. Radix Sort and Filter Bank are affected by an inefficiency in the reactor scheduler. As discussed in Section 5.2.1, the scheduler sorts the APG by level and executes reactions in the next level only after all reactions in the previous level have been completed. In these particular benchmarks, this simple strategy leads to a non-optimal execution, as some reactions are executed later than they could. A revision of the scheduling algorithm to improve the performance in these benchmarks remains for future work. The remaining parallelism benchmarks, however, highlight that LF can efficiently implement parallel algorithms. The LF implementations are on par with Akka and CAF and scale well with an increasing number of threads.

On average, LF outperforms both Akka and CAF. For 20 threads, the C++ runtime achieves a speedup of 1.85x over Akka and a 1.42x speedup over CAF. These speedups were calculated using the geometric mean over the speedups of individual benchmarks. Based on the presented results, we can conclude that LF can compete with and even outperform modern and highly optimized actor frameworks such as Akka and CAF. Particularly for workloads that require synchronization, LF significantly outperforms actor implementations. LF is as efficient as the actor frameworks in exploiting parallelism and scales well to a larger thread count. In summary, the deterministic concurrency provided by LF does not hinder performance but, instead, enables more efficient implementations. This is possible in part because the scheduler has insights into the program structure and explicit synchronization can be avoided in LF, as opposed to many of the original Savina actor benchmarks.

The performance comparison between C++ and Scala (Akka) needs to be taken with care, as other factors such as different library implementations and the behavior of the Java Virtual Machine (JVM) may influence performance. For instance, the large discrepancy between Akka and our implementation in the Pi Precision benchmark is explained by a less efficient representation of large numbers in Scala/Java. However, the other benchmarks of the Savina suite do not depend on external libraries and are designed to be more portable between languages. Also note that over all benchmarks, CAF only achieves an average speedup of 1.09x over Akka for 20 threads and is outperformed in 9 out of 16 benchmarks. For single-threaded execution, Akka outperforms CAF in 10 benchmarks and achieves an average speedup of 1.33x. This indicates that the implemented Scala workloads are comparable to the C++ implementations. Even considering a potential skew due to the JVM, our results clearly show that LF can compete with state-of-the-art actor frameworks.

To better understand the impact of the optimizations discussed in Section 5.2, Figure 5.8 also shows the speedup of our optimized runtime for 20 worker threads compared to a less optimized version. This baseline is an older version of the C++ runtime that is optimized in the sense that obvious bottlenecks were eliminated using common profiling and code optimization techniques, but that does not include the optimizations discussed in Sections 5.2.2 to 5.2.4. The average overall speedup (geometric mean) achieved by the optimizations is 2.18x. In particular, Big and Bank Transaction significantly benefit from the optimization for sparse communication patterns. The concurrency benchmarks (e.g., Concurrent Dictionary and Dining Philosophers), are mostly improved by reducing the contention on shared resources using lock-free algorithms. However, not all benchmarks benefit from the optimizations. The reduced performance in Ping Pong and Counting Actor shows that

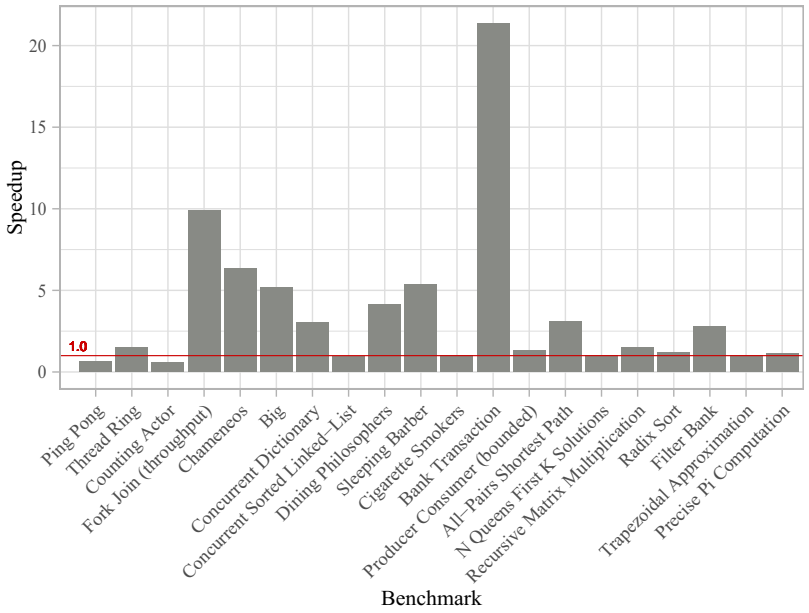


Figure 5.8: Speedup achieved by our optimized C++ runtime for 20 worker threads compared to an unoptimized version.

optimizing for efficient parallel execution also comes at a cost for simple sequential programs.

5.4 Conclusion

Unlike actors and related models for asynchronous concurrency, the reactor model enforces determinism by default, and features asynchronous behavior only when introduced deliberately. The presented evaluation based on Lingua Franca's C++ target, shows that the restrictions of the reactor model do not impede performance. On the contrary, LF achieves an average speedup of 1.85x over Akka and 1.42x over CAF. The LF C++ target combines reproducible (and testable) behavior with good performance. Moreover, the implementation of a wide range of benchmarks out of the Savina benchmark suite demonstrates that LF is indeed suitable for solving practical problems. The presented results underline the general applicability and the scalability of LF.

Partitioning Lingua Franca Programs

6

The previous chapter considered the scalability of LF and demonstrated that LF programs can achieve a high performance and even exceed the performance of actor frameworks for various problems. However, most of the Savina benchmarks used in this evaluation expose a regular structure that matches well with the simple level-based scheduling algorithm that Section 5.2.1 introduced. There is also a range of patterns for which our scheduling algorithm cannot fully exploit parallelism.

This chapter focuses on the known limitations of the scheduling mechanism introduced in Section 5.2. We discuss known patterns for which the scheduler cannot fully exploit parallelism or where the execution of some reactions is delayed longer than necessary. This chapter also proposes a solution which that on partitioning reactor programs into multiple segments that use independent schedulers but that need to use coordination strategies similar to federated execution (cf. Section 4.5).

6.1 Problem Analysis

This section discusses various examples that illustrate how the current scheduling algorithm may negatively impact performance and timeliness of execution.

6.1.1 Pipeline Parallelism

Consider the cascade of reactors shown in Figure 6.1. It consists of a source and a sink reactor, as well as three stages that perform some computation. Ideally, we would be able to exploit pipeline parallelism when executing this program. While each data item produced by the source is processed in sequential order by the stages, in principle, different stages could process different data items in parallel.

When using the scheduling algorithm discussed in Sections 3.1.4 and 5.2, however, the program is executed strictly sequentially. Figure 6.3 shows the corresponding timing diagram. This is because the reaction in each stage of the cascade depends on the reaction in the previous stage (cf. Figure 6.2). Furthermore, the scheduler only advances to the next tag once all reactions at the current tag have been executed. In other words, the scheduler imposes a global tag barrier, and different reactors may not operate at different tags.

In order to convert the cascade into a pipeline, such that the stages may execute in parallel, we can insert logical delays between the stages, as shown

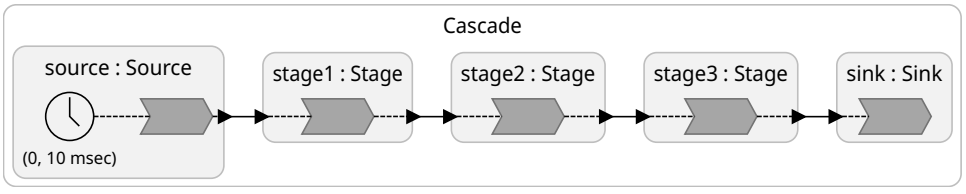


Figure 6.1: A simple cascade of reactors.

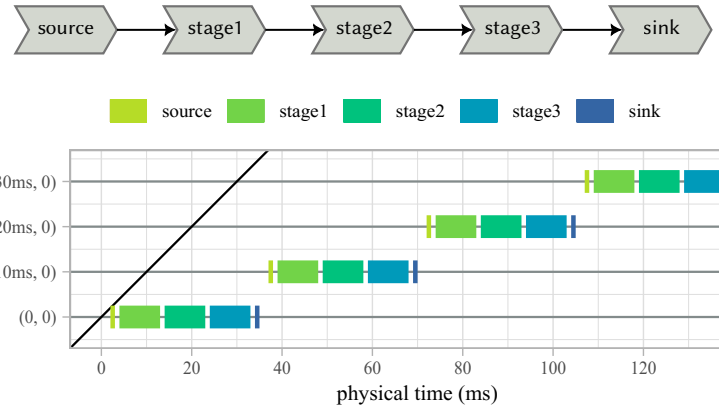


Figure 6.2: APG of the cascade program in Figure 6.1.

Figure 6.3: Timing diagram for the cascade program in Figure 6.1.

in Figure 6.4. Inserting logical delays breaks the dependencies between the pipeline stages (cf. Figure 6.5). By choosing a delay equal to the interval of the timer that triggers the production of new data items in the source reactor, we ensure that the events between stages are logically simultaneous.

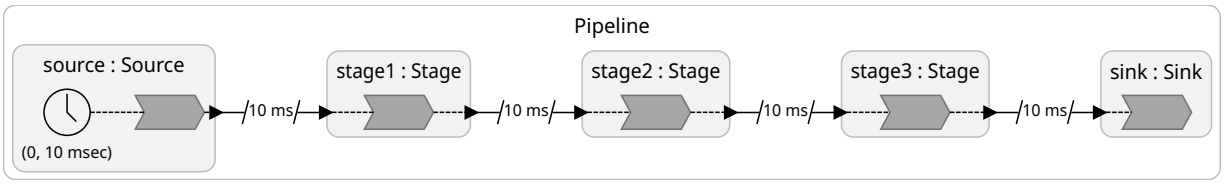


Figure 6.4: A simple reactor pipeline with delays in between stages.

Figure 6.6 shows the timing diagram for this pipeline application. At the initial tag $g_0 = (0, 0)$ only the source reactor is triggered by the timer. In the next tag $(10\,\text{ms}, 0)$, the source produces a new data item, and the first stage processes the previous data item in parallel. Starting from tag $(40\,\text{ms}, 0)$, all stages may execute in parallel, given that there is a sufficient amount of worker threads available.

While we can use logical delays to parallelize the execution of the program shown in Figure 6.1, inserting delays is not always desirable as it decreases consistency (cf. Section 4.5.4). Moreover, the described approach only works if the delays between pipeline stages are chosen carefully to always match the production rate, which is not always possible.

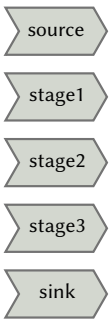


Figure 6.5: APG of the pipeline program in Figure 6.4.

6.1.2 Variability in Parallel Reactions

Most of the benchmarks considered in Section 5.3 use a regular program structure such that the workload is distributed evenly between parallel reactions. The workload performed by each reaction is quite uniform, and the variability in reaction execution time is negligible. However, when we increase the jitter in reaction execution time, e.g., because the complexity of the computation is data-dependent, then another problem arises.

Consider the program in Figure 6.7, which combines pipeline parallelism with data-level parallelism. In principle, `stage1`and `stage3`can execute in parallel to `stage2`and `stage4` However, due to the level sorting, the scheduler will only process `stage1`in parallel to `stage2`and `stage3`in parallel to `stage4` This is fine as long as the execution time between stages is evenly distributed and does not vary considerably.

Figure 6.8 shows a possible timing diagram for the example in Figure 6.7. For the first tag, all stage reactions take exactly 10 ms to execute. The reactions

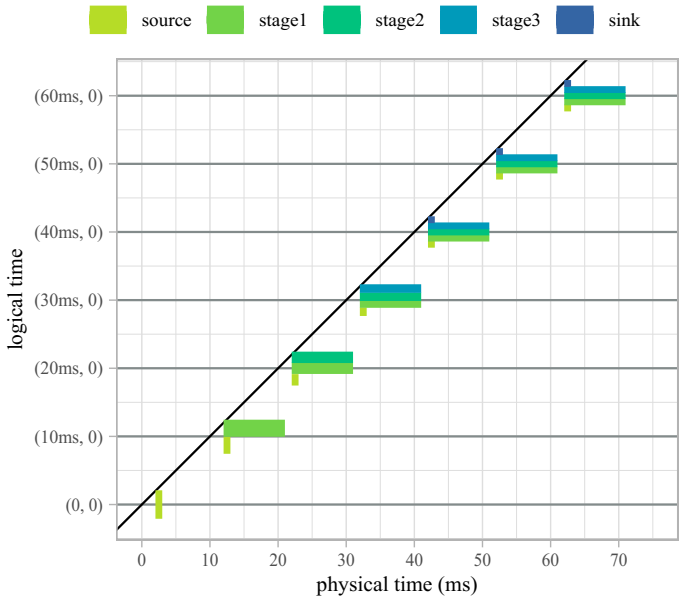


Figure 6.6: Timing diagram for the pipeline program in Figure 6.4.

in the upper part of the reactor diagram execute in parallel to the reactions in the lower part. However, for the tags (30 ms, 0) and (60 ms, 0), the execution time varies between stages. While one of the parallel stages takes 5 ms to complete execution, the other stage takes 15 ms. Due to the level-based execution strategy, the scheduler always waits until all reactions at the current level have completed executing before releasing the next batch of reactions for execution. This creates gaps in the schedule, which could in principle be avoided by executing reactions in subsequent stages earlier.

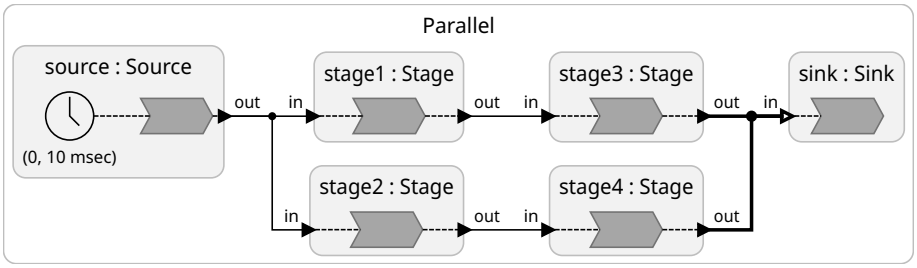


Figure 6.7: A program with two parallel sections.

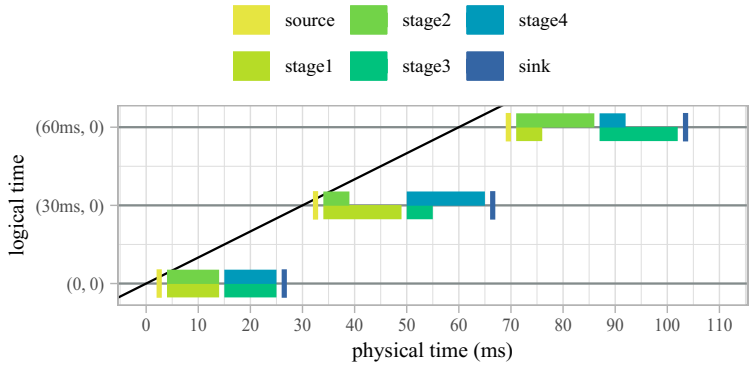


Figure 6.8: Timing diagram for the program in Figure 6.7.

6.1.3 Car Brake Example

Level-based scheduling and the global tag barrier not only limit the performance of some applications by missing opportunities for exploiting parallelism, but may also negatively impact the timing characteristics of safety-critical applications. Reconsider the simple car brake example shown in Figure 3.8 and discussed in Section 3.2.2 on Page 49. Figure 6.9 shows a slightly modified version.

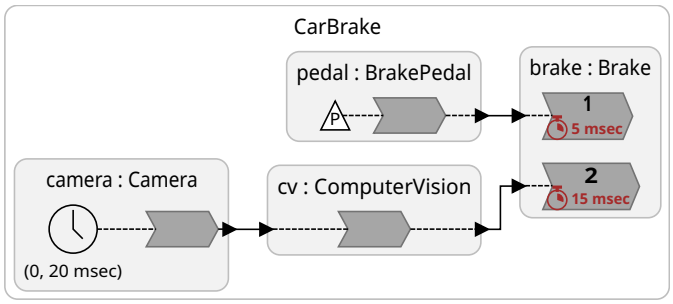


Figure 6.9: A simplified brake assistant with deadlines.

The program consists of a brake actuator that receives signals from a physical brake pedal as well as from a computer vision component. The computer vision component receives frames from a camera at an interval of 20 ms. It analyzes the frames and tries to identify objects in the vehicle's path. If such an object comes too close, `ComputerVision` sends a signal to the brake actuator to apply the brakes. The `Brake` reactions are annotated with deadlines. The maximum lag for applying the brakes after the pedal is pressed should be 5 ms. The brake reaction that reacts to messages coming from `ComputerVision` has a larger deadline of 15 ms. This is to account for the execution time of the computer vision algorithm, which we assume to be about 10 ms.

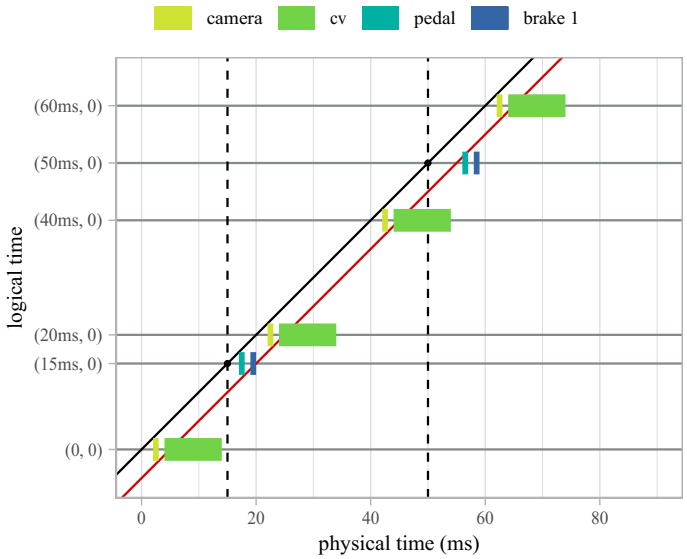


Figure 6.10: A timing diagram showing one potential execution of the simplified brake assistant application in Figure 6.9.

The timing diagram in Figure 6.10 visualizes one potential execution trace of the brake assistant program. The diagonal red line denotes the deadline of the first brake reaction (i.e., the reaction handling messages from the pedal). The vertical dashed lines denote the reading of physical time at the moment when the brake pedal is pressed. Pressing the brake pedal injects new events into the event queue by scheduling the physical action.

Due to the timer, the camera creates new events at intervals of 20 ms. At each

triggering of the timer, the camera reaction executes and sends the frame to `ComputerVision`where it is processed by the reaction. In this example, the computer vision algorithm does not detect any dangerous obstacles and hence does not send any signals to the brake. However, at 15 ms after startup, the brake pedal is pressed, which creates the event at tag 15 ms, 0. This event is handled immediately by the scheduler, and both the pedal and the brake reaction execute within the deadline.

50 ms after startup, the brake pedal is pressed again for the second time. In this case, the newly created event with tag (50 ms) cannot be processed immediately, as the runtime is currently still executing the computer vision reaction at tag (40 ms). Only once this reaction is processed completely, the scheduler advances to the next tag and handles the brake pedal event. However, this is too late, considering the deadline.

A long-running reaction in one part of the program might block advancement in seemingly unconnected parts of the program. When assigning deadlines to the brake reactions, we need to account for potential interference between the computer vision and the brake pedal reactions. This implies, however, that we potentially need to choose large deadlines, which might be infeasible for achieving certain safety requirements.

6.2 Partitioning with Enclaves

The previous section illustrated different problems that arise from the fact that the level-based scheduler executes some reactions later than it could. This may lead to interference between reactions that do not depend on each other. There are two options for addressing this problem.

First, we can improve the scheduling algorithm and replace the level-based scheduling and the global tag barrier with more elaborate algorithms that can handle different reactors executing at different tags and are capable of identifying reactions that are ready for execution without relying on levels. However, deriving better scheduling algorithms so that they can also be implemented efficiently has proven challenging.

Alternatively, we can try to isolate independent reactions and partition programs so that we can use multiple schedulers. Federates provide one mechanism for partitioning LF programs. Each federate has its own scheduler and operates independently, while also coordinating with the RTI or other federates. However, federates also imply execution in separated address spaces. Federated execution is a viable solution for partitioning distributed applications, but it incurs significant overhead if distributed execution is not required.

Portioning LF programs, however, does not require distributed execution and can also be done within the same address space. We call such a local partition *scheduling enclave*. In LF, an enclave is a reactor that executes separately from the rest of the system. Each enclave has its own scheduler and pool of worker threads. To preserve determinism when programs are partitioned into multiple enclaves, we need to utilize coordination strategies similar to those used for federated execution (cf. Section 4.5). In Lingua Franca, an enclave can be created by annotating an arbitrary reactor instantiation with the `@enclave`attribute.[1] This slightly changes the output of the code generator. Instead of generating a contained reactor instance, the code generator creates a new environment (cf. Section 4.7.1) and assigns the enclave reactor instance as a top-level reactor. The code generator further adjusts the incoming and outgoing connections of the reactor instance so that they correctly implement

1: LF includes an attribute syntax. Each attribute is prefixed by the `@` symbol and may be followed by a list of argument assignments enclosed in parentheses. The attribute syntax was omitted from the LF syntax definition in Listing 4.2.

cross-enclave communication. The following section discusses how such connections can be implemented.

6.3 Coordinating Enclaves

In principle, we can use the two strategies currently available for coordinating federates (cf. Section 4.5.4) to coordinate the execution of multiple enclaves. However, both strategies have significant limitations. The decentralized coordination strategy requires WCET estimates, which are typically available for hard real-time applications but not for general-purpose applications. The centralized coordination strategy does not require additional information but instead uses a central coordinator, which also imposes a considerable bottleneck.[2]

2: Compared to federated execution, the overhead induced by the RTI for enclaves would be reduced, as all communication happens within the same address space, but still, the central coordination limits scalability.

Since both coordination strategies are not ideal, the current C++ implementation for enclaves described in this chapter attempts to find a middle ground between them. In contrast to federated execution, coordinating enclaves removes the complexity of networked communication. Therefore, the implementation of coordination strategies for enclaves presents a fertile ground for exploring new approaches. New strategies can be prototyped without implementing a complete network stack, and debugging and analyzing the execution is significantly easier than debugging distributed applications. The following presents initial work in implementing enclaves in the C++ runtime and deriving a new coordination scheme.

6.3.1 Generalizing Logical Time Synchronization

First, we introduce the concept of time barriers in the C++ runtime. As discussed in Sections 3.1.4 and 4.4.2, the runtime implements a physical time barrier that ensures that logical time does not progress faster than physical time. In the scheduling algorithm shown in Listing 3.3, synchronization with physical time is an integral part of the algorithm. However, we can also implement such synchronization mechanisms as separate, generalized components.

The C++ runtime provides two classes called `PhysicalTimeBarrier` and `LogicalTimeBarrier` that may be used to control how far a program may advance in logical time. The simplified interface of both classes is shown in Listing 6.1. These barriers are designed in analogy to counting semaphores. Semaphores are typically used to coordinate access to a shared resource between multiple threads. A thread may *acquire* the semaphore to obtain access to a resource and *release* the semaphore once it is done. If the resource is not available, the acquire operation typically blocks until another thread releases the semaphore.

The time barriers are used similarly to semaphores. However, they do not manage access to a countable resource, but instead control the advancement of logical time. Thus, in a sense, the time barriers manage the resource logical time. A scheduler that intends to advance logical time to the next tag first acquires the tag on the barrier. This blocks until a concurrent process releases the tag. Note that the `acquire_tag` methods shown in Listing 6.1 accept a function that may abort the waiting when it returns true. This mechanism is required for reacting to external modifications to the event queue. For instance, a physical action could be scheduled at an earlier tag while the scheduler tries to acquire a later tag.

```
class PhysicalTimeBarrier {
 public:
  static auto try_acquire_tag(const Tag& tag) -> bool;
  static auto acquire_tag(
    const Tag& tag,
    const std::function<bool(void)>& abort_waiting) -> bool;
};

class LogicalTimeBarrier {
  Tag released_tag;
 public:
  void release_tag(const Tag& tag);
  auto try_acquire_tag(const Tag& tag) -> bool;
  auto acquire_tag(
    const Tag& tag,
    const std::function<bool(void)>& abort_waiting) -> bool;
};
```

Listing 6.1: Time barrier classes as provided by the C++ runtime.

The `PhysicalTimeBarrier` synchronizes with physical time. It releases automatically as the local physical clock advances. Therefore, it does not expose an explicit release operation. The `LogicalTimeBarrier`, however, is intended for coordinating the progress of logical time between two enclaves. While a downstream enclave acquires the barrier before starting to process a tag, an upstream enclave releases the barrier to indicate that it will not produce new events with a tag lower or equal to the released tag.

6.3.2 Generalizing Program Inputs

To effectively utilize the concept of generalized time barriers in the C++ runtime, we also need to generalize the concept of external inputs that originate from concurrent processes. Such external inputs may include physical actions as well as incoming connections that originate from other enclaves or federates. To decide whether it is safe to process a given tag, the scheduler needs to consider each of the external inputs and infer whether they could still produce events with a lower or equal tag. To generalize this, the C++ runtime introduces the concept of *input actions*.

Consider the UML diagram in Figure 6.11. It visualizes the inheritance relation for all classes that inherit from the `Action` class. Compared to the diagram shown in Figure 4.17, the `BaseAction` class is extended by another method called `acquire_tag` which must be implemented by every child class. Before processing a tag, the scheduler calls `acquire_tag` on all input actions. In Figure 6.11 all action classes that are considered input actions are highlighted in dark gray. They automatically register themselves in the local environment on instantiation.

`PhysicalAction`, for instance, is an input action, and its implementation of `acquire_tag` internally uses a `PhysicalTimeBarrier`. Thus, any program that has a physical action may not advance logical time faster than physical time progresses. This has the additional benefit that programs with physical connections always behave correctly, even if fast mode is enabled.

To model connections between enclaves, we also introduce an abstract `Connection` class that extends `Action`. Enclave connections are similar to actions in that they may schedule new events, are typed, and may carry a value. The `Connection` class implements some common functionality, e.g., binding to ports, and also defines pure virtual methods that need to be implemented by the concrete connection classes to provide the correct functionality.[3]

3: These methods and other details are omitted from the overview in Figure 6.11.

Since we introduce an abstraction for connections, we can also use this for implementing physical connections and after delays natively in the C++

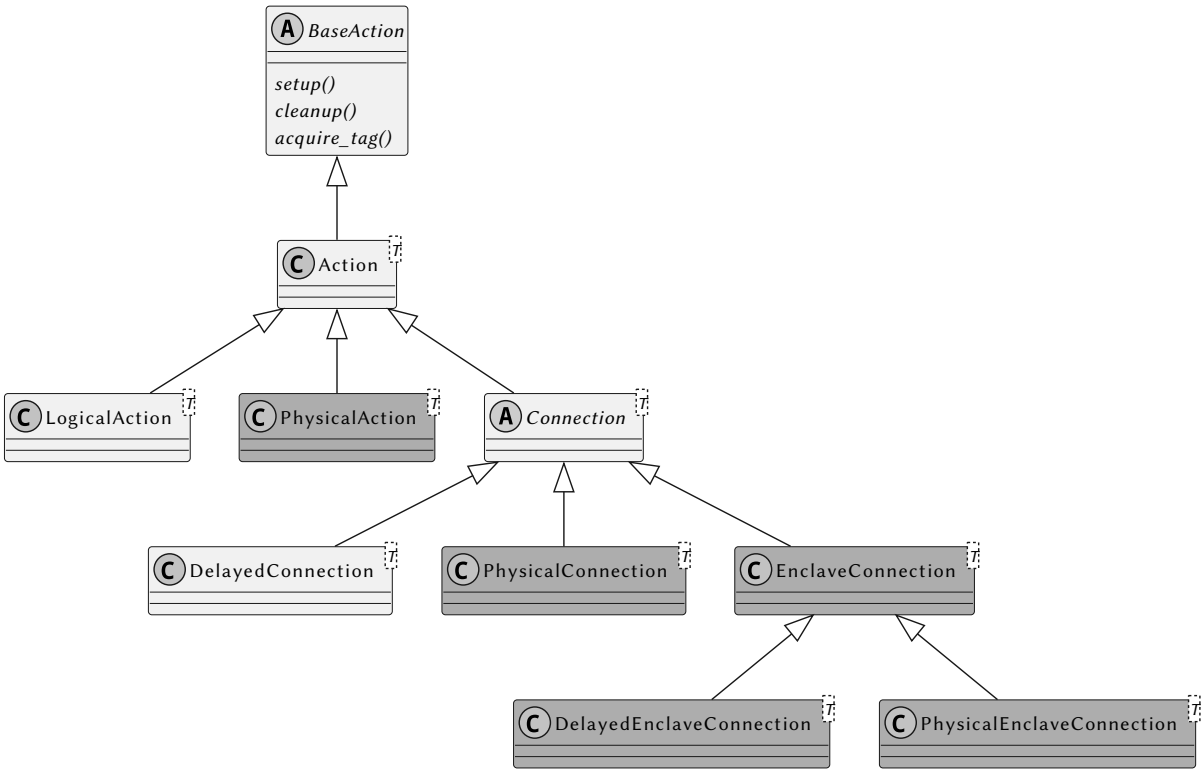


Figure 6.11: UML diagram showing the inheritance relation between actions and various connection classes in the C++ runtime.

runtime instead of a transformation in the LF compiler (cf. Section 4.4.5). The class `DelayedConnection` implements after delays. It can be considered a logical action that can be scheduled from another reactor by setting a port. Similarly, `PhysicalConnection` implements physical connections and can be considered a physical action that can be scheduled from another reactor via a port. All physical connections are also input actions, as they behave similarly to physical actions. Essentially, physical connections relay asynchronous (i.e., untagged) messages that the program sends to itself. Note that there is no representation for regular connections (i.e., connections between two ports within the same environment), as those do not create new events at later tags and are not analogous to actions.

The class `EnclaveConnection` implements connections between two enclaves. When the upstream enclave sets the connected port, then the connection schedules a new event at the connected downstream enclave. The newly created event has the same tag as the current tag of the upstream enclave when setting the port. In addition, the connection contains a logical time barrier, which is used to coordinate the execution between enclaves. `DelayedEnclaveConnection` and `PhysicalEnclaveConnection` are additional specializations that implement after delays and physical connections between enclaves. All enclave connection classes are input actions, as they allow scheduling events from other concurrently executing enclaves.

The newly introduced abstractions unify the mechanism used for scheduling new events and provide a common interface for managing the inputs of each enclave.

6.3.3 Coordination

The C++ runtime implements a point-to-point coordination scheme for enclaves. Instead of consulting a central coordinator, each enclave consults its direct neighbors using the connection classes and time barriers discussed above. Whenever an enclave completes processing a tag, it notifies all outgoing connections. Before processing a new tag, each enclave calls `acquire_tag` on all of its input actions, which includes all enclave connections.

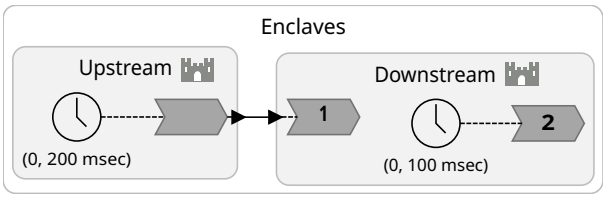


Figure 6.12: An example program with two enclaves.

Consider the example program shown in Figure 6.12. It consists of two reactors called `Upstream` and `Downstream`. Both reactors are enclaves, which is indicated by the castle symbol.[4] The connection is implemented using the `EnclaveConnection` class, which internally uses a logical time barrier.

4: The castle symbol is licensed under CC BY-SA 4.0 Deed and attributed to Wikipedia user Douglal.

The upstream enclave has a timer that triggers at intervals of 200 ms. When the upstream enclave finishes processing a tag, it notifies the outgoing connection. This releases the tag on the logical time barrier that the connection keeps internally. For each tag that the downstream enclave processes, it calls `acquire_tag` on the connection object, which in turn calls `acquire_tag` on the internal logical time barrier. Thus, `Downstream` always waits until `Upstream` releases a tag before processing it. This effectively ensures that a downstream enclave only processes events once it is certain that no upstream enclave will send any messages with an earlier tag.

Figure 6.13 shows the timing diagram for the example in Figure 6.12 using the protocol described above. In addition to the scheduled events and executed reactions, the diagram also shows when `Downstream` acquires a tag and when `Upstream` releases a tag. While the protocol described above enforces a deterministic execution, it does not ensure a timely execution. At tags (100 ms, 0) and (300 ms, 0), the reaction triggered by `Downstream`'s timer is executed with a significant lag. This is because the upstream enclave only releases a tag after it has completed processing it. Thus, the downstream enclave does not receive any notifications for tags for which there is no event scheduled in the upstream enclave. As `Upstream` has no event scheduled at (100 ms, 0), `Downstream` has to wait until it receives the release for the tag (200 ms, 0) before it can process the tag (100 ms, 0).

In the extreme case, the upstream enclave could have an empty event queue, and downstream enclaves would wait indefinitely for a release. To ensure that a downstream enclave may progress even if an upstream enclave has no events or only sparse events, we deploy a simple mechanism. When a downstream enclave tries to acquire a tag that has not been released yet and the upstream has no event scheduled at this tag, we insert an empty event at this tag in the event queue of the upstream enclave. An empty event denotes an entry in the event queue at a specific tag without any associated actions or other triggers that are present at this tag.

By inserting an empty event, we ensure that the upstream enclave processes the requested tag. It will acquire the tag from all its input actions and, since the event is empty and no reactions are triggered, immediately release the tag. Figure 6.14 shows the timing diagram for this extended coordination protocol. This protocol ensures both determinism and timeliness of execution.

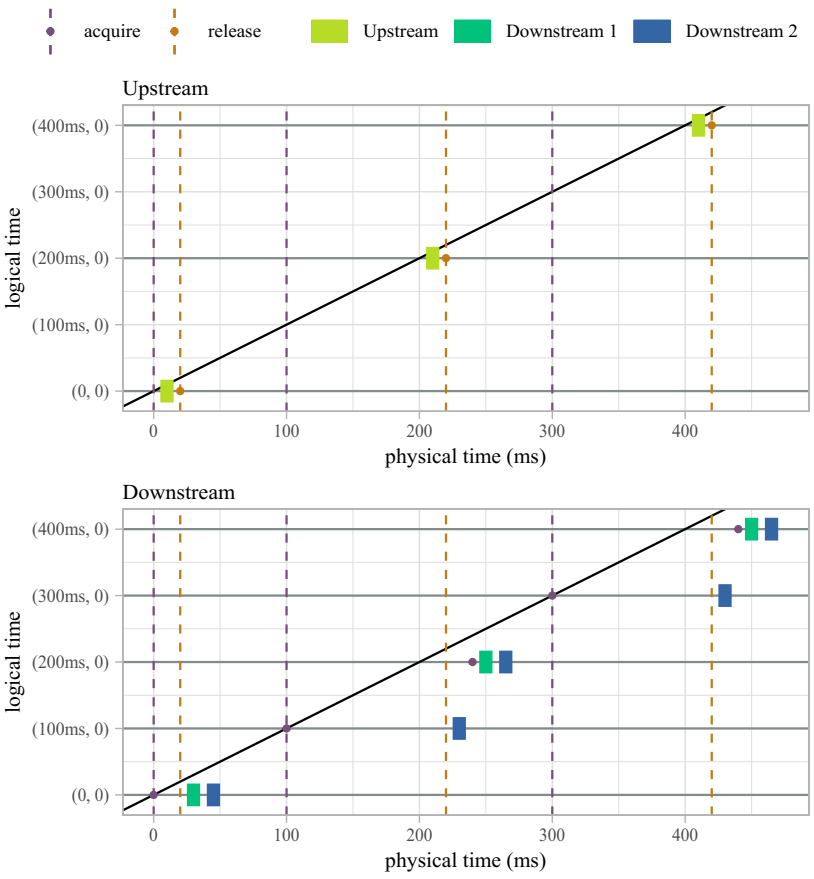


Figure 6.13: Timing diagram for the example in Figure 6.12 using a naive coordination scheme.

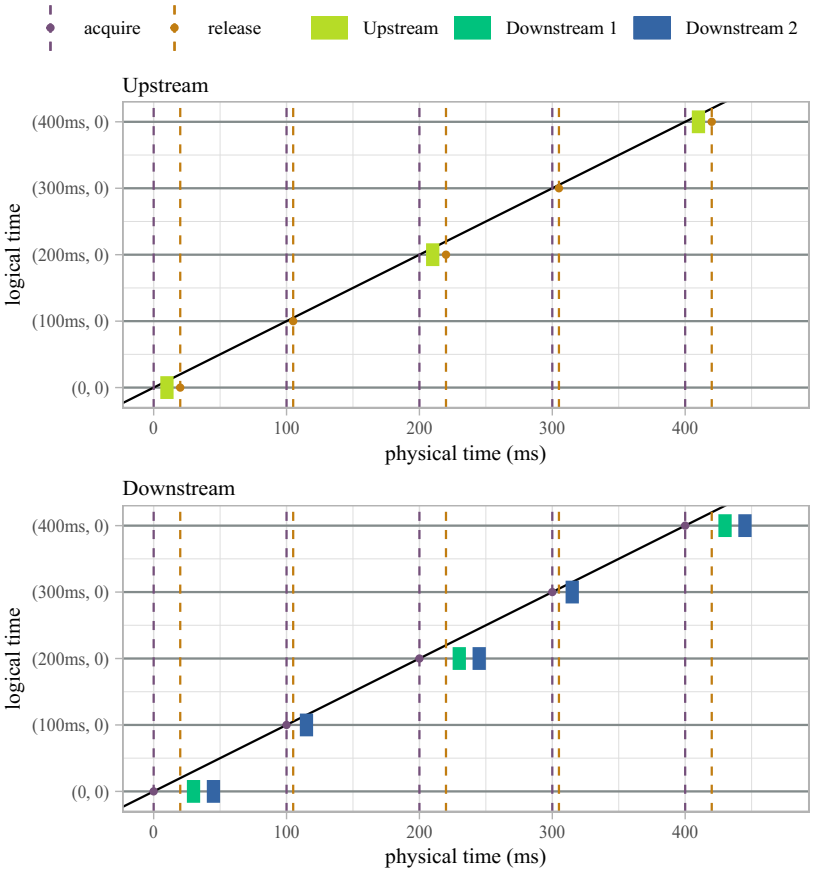


Figure 6.14: Timing diagram for the example in Figure 6.12 using the complete coordination scheme that inserts empty upstream events.

We can further optimize the protocol by allowing enclaves to peek ahead in their event queues. An enclave that has no input actions can release the largest possible tag $g_r \in \mathbb{G}$ that is strictly less than the tag of the next event in the event queue ($g_r < g_{\text{next}}$). If the enclave has input actions, we can also do this transitively and release the minimum of g_r and all the tags released by the input actions.

The protocol described above is implemented by the `EnclaveConnection` class. If there is an after delay annotated on a connection between enclaves, then the code generator inserts a `DelayedEnclaveConnection` instead. The behavior of `DelayedEnclaveConnection` with a delay d is similar to `EnclaveConnection`, but it differs in two key aspects. First, when the upstream enclave sends a message at tag g, the message is inserted in the downstream enclave's event queue at $\mathscr{D}(g, d)$. Second, when the downstream enclave acquires a tag g_a, then we first subtract the delay d before acquiring the tag from the internal logical time barrier.

Subtracting a delay from a tag is subtle as the delay function $\mathscr{D}$ is not injective. For instance:

$$\mathscr{D}(10\,\text{ms}, 0, 5\,\text{ms}) = \mathscr{D}(10\,\text{ms}, 1, 5\,\text{ms}) = (15\,\text{ms}, 0)\,.$$

Therefore, the function $\mathscr{S}$ that is used for subtracting a delay d from a tag g determines the largest possible tag that, when delayed by d, results in g. Let M denote the largest possible microstep value. $\mathscr{S}$ is defined as follows:

$$\mathscr{S}(t, m, d) = \begin{cases} (t, m-1) & \text{for } d = 0 \wedge m > 0 \\ (t-d, M) & \text{for } d > 0\,. \end{cases}$$

When the downstream enclave acquires the tag g on a connection with delay d, then the connection internally acquires the tag $\mathscr{S}(g, d)$ from its barrier. Note that $\mathscr{S}$ is not defined for $d = 0$ and $m = 0$. A connection that imposes a microstep delay may never produce an event with the tag $(t, 0)\,\forall t \in \mathbb{T}$.

The class `PhysicalEnclaveConnection` is inserted when a connection between enclaves is physical. Internally, it uses a `PhysicalTimeBarrier` instead of a logical barrier. Also it uses a physical action to assign a new tag when a message is received. This fully decouples the execution of the downstream enclave from the upstream enclave. The barrier is automatically released as physical time advances and the execution of the downstream enclave is only constrained by the advancement of physical time.

The protocol introduced in this section is not limited to coordinating enclaves and can also be applied for coordinating federates. The `Connection` class introduced in Section 6.3.2 abstracts over the implementation details of the underlying communication mechanism. We can provide different implementations that assume a shared address space, as is the case for enclaves, or that use a networked communication layer to bridge across address spaces.

6.4 Examples

This section reconsiders the motivational examples introduced in Section 6.1 and highlights how we can use enclaves to solve the identified problems.

6.4.1 Pipeline Parallelism

In the pipeline example (cf. Figure 6.1), we can simply mark each pipeline stage as an enclave. This decouples the execution between the stages, and each stage may execute its internal reaction as soon as the predecessor releases the corresponding tag. The timing diagram for the version of the pipeline example using enclaves is shown in Figure 6.15 with a separate diagram for each of the enclaves. To better visualize the pipeline parallelism that the enclave version exploits, Figure 6.16 also shows a combined diagram.

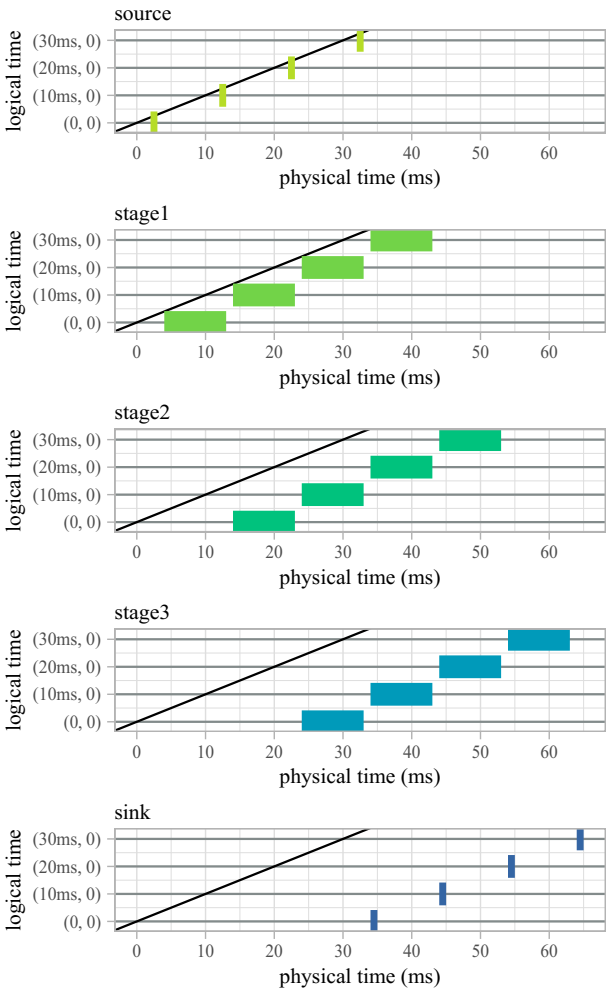


Figure 6.15: Timing diagram for the pipeline example in Figure 6.1 using enclaves for each of the stages.

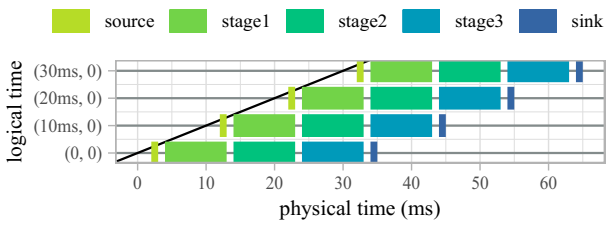


Figure 6.16: Timing diagram that combines the diagrams for the individual enclaves in Figure 6.15.

6.4.2 Variability in Parallel Reactions

In the example shown in Figure 6.7, we can also simply mark all reactors as enclaves to circumvent the barriers imposed by the level mechanism. Figure 6.17 shows the combined timing diagram for the version using enclaves. In this version, the two branches may operate independently. `stage3`only needs to wait for the release message from `stage1`and `stage4`only needs to wait for the release message from `stage2` If exploiting pipeline parallelism between subsequent stages is not required, they could also be combined into a single enclave. This can be done by introducing two wrapper reactors that are marked as enclaves, one of which containing `stage1`and `stage3`and the other containing `stage2`and `stage4`

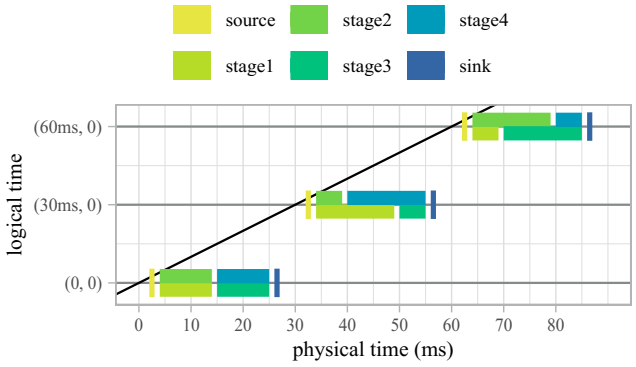


Figure 6.17: Timing diagram for the example in Figure 6.7 using enclaves for each of the reactors.

6.4.3 Car Brake Example

Using enclaves for fixing the car brake example in Figure 6.9 so that we can avoid deadline violations is more subtle. If we simply mark all reactors as enclaves, we are still at risk of missing deadlines. This is because `Brake`needs to wait for a tag release both from `BrakePedal`and `ComputerVision`. In the worst case, `ComputerVision`just started executing an earlier tag, and thus `Brake`might need to wait 10 ms until it receives a release. We can resolve this by introducing a logical delay between `ComputerVision`and `Brake`

Figure 6.18 shows a version of the brake example that uses enclaves. This implementation introduces a new enclave reactor called `BrakeAssistant` that contains both `Camera`and `ComputerVision`. Thus, both reactors are contained within the same scheduling enclave. The logical delay of 12 ms accounts for the WCET of the computer vision algorithm as well as additional overhead for capturing the camera frame and scheduling reactions. This version of the car brake example specifies that the `Brake`'s view of the system state be consistent with the view of `Camera`and `ComputerVision` from 12 ms ago. This approach is similar to LET and can be explained with

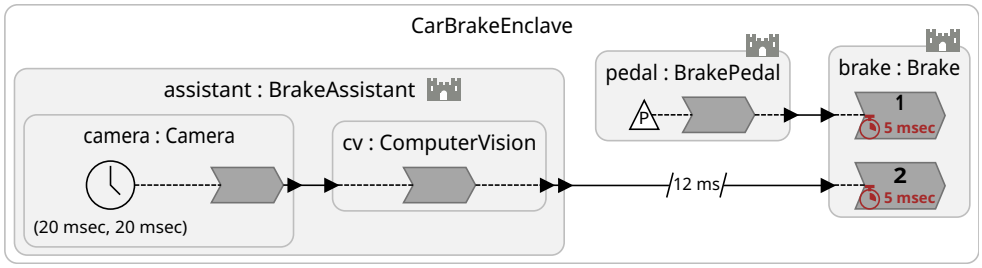


Figure 6.18: A modification of the car brake example in Figure 6.9 that uses enclaves and a logical delay between the assistant and the brake.

the CAL theorem. To improve the availability of the brake and meet the deadlines, we need to accept a certain amount of inconsistency.

One can argue, however, that the version in Figure 6.18 does not fully solve the problem. It enforces a deterministic ordering between the brake signal coming from `BrakePedal` and the one coming from `BrakeAssistant`. Although we were able to reduce the time `Brake` might have to wait, it still requires a release message from `BrakeAssistant`. If the execution of the computer vision algorithm takes longer than expected, this also affects the brake's ability to react within bounded a time to a signal from the pedal.

In fact, determinism and consistency are neither required nor desired in this example. To fulfill safety requirements, the `Brake` reactor should actuate the brakes as soon as it receives a message that indicates it should do so. In this scenario, it is not necessary for the brake to have a consistent view of the system's state. When the brake pedal is pressed, it is irrelevant that `ComputerVision` might produce a message with an earlier tag.

We can use physical connections, as shown in Figure 6.19, to fully decouple `Brake` from the rest of the system. The physical connections assign new tags in the order of message reception. Thus, the `Brake` reactor will immediately process incoming messages in their order of arrival. The behavior of the `Brake` reactor in this example is equivalent to a Hewitt actor.

The car break example illustrates how LF programmers can control the system's behavior. Logical delays enable a trade-off between availability and consistency. In cases where availability is favored over consistency, programmers can introduce physical connections to fully decouple the execution of reactors similar to the Hewitt actor model.

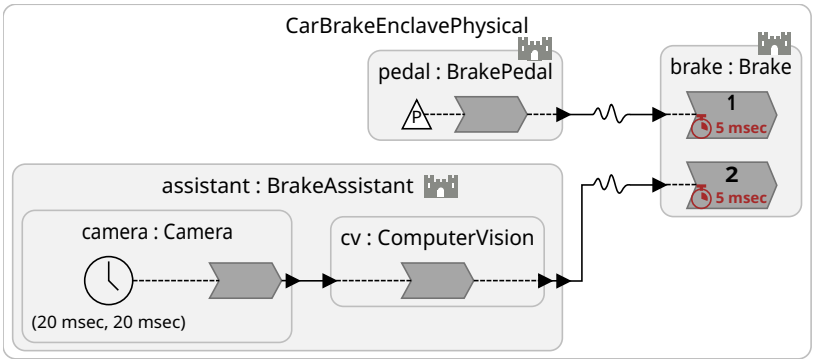


Figure 6.19: A modified version of the car brake example in Figure 6.9 that uses enclaves and physical connections.

6.5 Enclave Patterns

The car brake example discussed in the previous section illustrates that we can use enclaves to model the semantics of other MoCs like LET or Hewitt actors in LF. This section further develops this idea and presents design patterns that leverage the capability of enclaves to execute independently of the rest of the system.

6.5.1 Hewitt actors

The semantics of Hewitt actors can be reproduced in LF by creating an enclave where all inputs are connected via physical connections. We have used this pattern previously to decouple the brake in Figure 6.19 from the rest of the system. Figure 6.20 generalizes this pattern. The inner reactor `Behavior`

models the behavior of the Hewitt actor. It may contain arbitrary state variables and reactions. The outer reactor, called `HewittActor`, instantiates the behavior reactor and marks it as an enclave. It further forwards all the outputs of `Behavior` to its own outputs and connects all inputs via physical connections.

Using this pattern, we can put an arbitrary reactor in place of `Behavior`. This pattern effectively converts the inner reactor into a Hewitt actor. In order to fully match the semantics of Hewitt actors, the inner reactor should not contain any timers or logical actions. However, it may use physical actions to send untagged messages to itself.

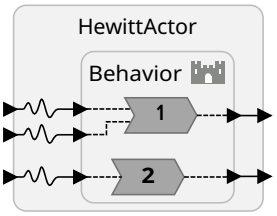


Figure 6.20: A Hewitt actor implemented with enclaves in LF.

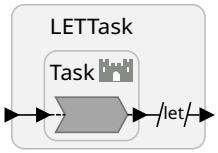
6.5.2 LET Tasks in LF

As discussed previously, using a logical delay in Figure 6.18 makes this solution of the car brake example similar to LET. We can generalize this approach and define a pattern for implementing LET tasks in LF. In fact, we can define a generic library reactor that provides LET semantics.

Listing 6.2: A generic LET task implemented with enclaves in LF.

```
target Cpp

reactor LETTask<T, U>(
  task: {= std::function<U(T)> =} = {=nullptr=},
  let: time = 0
) {
  input in: T
  output out: U

  @enclave
  t = new Task<T, U>(task=task)

  in -> t.in
  t.out -> out after let
}
```

```
reactor Task<T, U>(
  task: {= std::function<U(T)> =} = {=nullptr=}
) {
  input in: T
  output out: U

  reaction(in) -> out {=
    out.set(task(*in.get()));
  =}
}
```



Consider the code and diagram in Listing 6.2. This defines a library reactor called `LETTask` that has one input and one output port. The reactor is generic and defines two type parameters, `T` and `U`. `T` denotes the type of the input port, and `U` denotes the type of the output port. The reactor is further parameterized by a function that is expected to implement the task's functionality and map `T` to `U`. The second parameter denotes the task's LET.

The actual task is implemented in an inner reactor called `Task`. `Task` is defined similarly to `LETTask`. However, it defines a reaction that applies the task function to the received input and writes the result to the output.

`LETTask` instantiates `Task` as an enclave and forwards both the type parameters and the task function. In addition, `LETTask` directly forwards the input port to `Task`. The output produced by `Task` is forwarded to the outer output, but with a logical delay of precisely `let`.

This library reactor can be used to model arbitrary LET applications in LF. Thus, LET is a subset of reactors and Lingua Franca. The reactor model, however, is strictly more general as it exposes the full trade-off between fully synchronous and LET semantics.[5]

5: E. A. Lee and Lohstroh 2022, *Generalizing Logical Execution Time*.

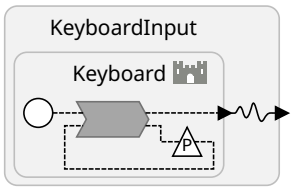
6.5.3 Input Reactors

Modeling system input is another problem that can be conveniently solved with enclaves. The reactor model defines physical actions for modeling system input, but does not specify precisely how physical actions can be scheduled in an external context. Consider, for instance, a reactor program that reads keyboard inputs and reacts to them.

In C/C++, we can use the function `getchar()` to read one character from the keyboard input stream, like it is done in the Reflex Game example (cf. Listing 4.17). However, this is a blocking operation. It waits until a key is actually pressed. Calling a blocking function from a reaction would mean that the entire program execution might hold until the reaction completes. This is not acceptable for interactive applications that constantly update the system state and output independent of when user input arrives. We can solve this by spawning a thread at startup that repeatedly calls `getchar()` in a loop and schedules a physical action when `getchar()` returns a value. This solution, however, is inelegant due to the pitfalls associated with threads.

A more elegant solution uses enclaves. Consider the pattern for modeling keyboard inputs in Listing 6.3. The `Keyboard` reactor defines an output port and a physical action called `next`. This reactor is instantiated as an enclave within the `KeyboardInput` reactor, and its output is forwarded via a physical connection. This fully decouples the execution of the `Keyboard` reactor from the rest of the system.

```
target Cpp

reactor Keyboard {
  output key: int
  physical action next

  reaction(startup, next)
    -> next, key
  {=
    int c = getchar();
    key.set(c);
    if (c != EOF) {
      next.schedule();
    }
  =}
}

reactor KeyboardInput {
  output key: int

  @enclave
  k = new Keyboard()

  k.key ~> key
}
```



Listing 6.3: A keyboard input reactor implemented with enclaves in LF.

The reaction defined by `Keyboard` is triggered by ***startup*** or `next`, and it may schedule `next` and set the output port. The reaction body directly calls `getchar()`. Since `Keyboard` executes completely independently of the rest of the system, it can use a blocking call. This only stalls the execution of the `Keyboard` enclave until a key is pressed. Once `getchar()` returns, the pressed key is forwarded to the output port. Since `KeyboardInput` forwards the key press using a physical action, a new event with a tag based on the current reading of physical time will be inserted in the receiving reactors.

To trigger reading the next key, the reaction body also schedules the physical action `next`. This creates a new event with a tag based on the current reading of time, which will be handled immediately. In principle, we could also use a logical action for `next`. However, it would be difficult to specify a meaningful logical delay. Using a physical action has the benefit that the logical delay between reaction executions is automatically aligned to the physical time that passed while the reaction was blocked.

6.6 Limitations

While the current implementation shows promising results, there are also several limitations that need to be addressed. This section discusses the most important limitations and sketches how they could be resolved in future work.

6.6.1 Cycles

The examples presented in Section 6.4 illustrate the effectiveness of enclaves and the underlying coordination scheme for solving real-world problems. However, all the previously discussed examples are acyclic. While the coordination scheme in principle also works for programs with cyclic structures, it can become inefficient.

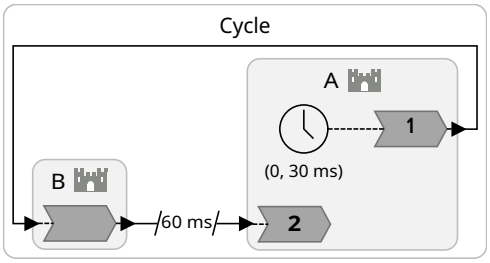


Figure 6.21: An example program with a cycle between two enclaves.

Consider the example program in Figure 6.21. It consists of two enclaves, A and B, where A sends a message to B in intervals of 30 ms and B sends a message back to A with a logical delay of 60 ms. Figure 6.22 shows a sequence diagram that illustrates the messages exchanged between the two enclaves and the connection objects.

Right after the enclaves start execution, only A has an event scheduled at the startup tag $g_0 = (0, 0)$. A tries to acquire this tag from the incoming connection [1]. This succeeds immediately [2], as the connection is delayed and each connection automatically releases the largest possible tag that is strictly less than g_0 when constructed. A processes its first reaction, which sets its outgoing port [3] and thereby schedules a new event in B at tag $(0, 0)$. B tries to acquire this tag [5], which succeeds as soon as A releases the tag [6, 7]. B's reaction also sets its outgoing port [8], which schedules a new event in A [9]. Since the connection from B to A is delayed by 60 ms, this new event is scheduled at tag $(60\,\text{ms}, 0)$. Finally B releases the tag $0, 0$ [10].

The steps from [11] to [20] and also from [21] to [30] repeat this sequence. In steps [11] and [12], the acquire operation succeeds immediately because $(30\,\text{ms}, 0)$ minus a delay of 60 ms is still before the startup tag g_0. In steps [21] and [22], the acquire operation succeeds because B has already released the tag $(30\,\text{ms}, 0)$. When A processes tag $(30\,\text{ms}, 0)$, it executes both reactions because the timer and the input from B are present.

The sequence diagram shows that programs with cycles between enclaves can, in principle, be executed using the presented coordination scheme. However, this only works efficiently as long as the sum of delays on the connections along the loop is equal to or larger than the interval in which events are processed.

Consider the sequence diagram in Figure 6.23, which shows the execution of the same example program, but with a 10 ms delay instead of the 60 ms in the original version. Messages [1] to [7] are identical to the first messages in Figure 6.22 and are therefore omitted from the figure. Messages [8] to [10]

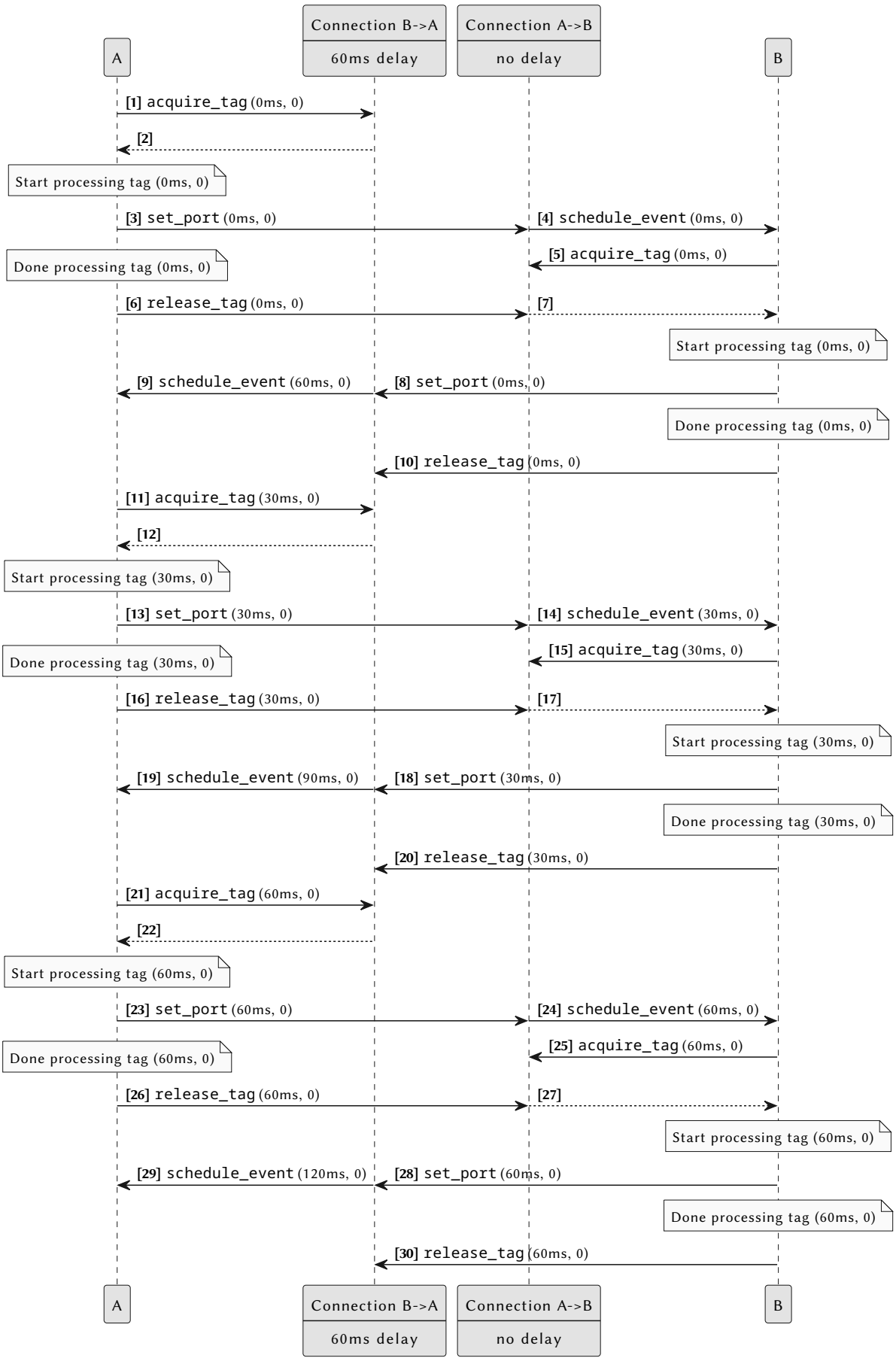


Figure 6.22: A sequence diagram that shows the communication between A and B for the program in Figure 6.21.

are also similar, but the new event in A is scheduled at tag $(10\,\text{ms}, 0)$ [9] due to the lower delay on the connection.

A acquires the tag $(10\,\text{ms}, 0)$ from the incoming connection. Since the connection is delayed by 10 ms, the connection tries to acquire the tag

$$\mathcal{S}(10\,\text{ms}, 0, 10\,\text{ms}) = (0, M)$$

from its internal barrier. As B has not released this tag yet, the connection schedules an empty event in B at this tag [12]. This ensures that B processes the tag and releases it once it can guarantee that it will not receive any earlier events. B, in turn, tries to process the newly scheduled event and also acquires the tag $(0, M)$ from its incoming connection [13]. Since A has not released the tag $(0, M)$ yet, this schedules an empty event in A [14]. Acquiring this tag succeeds immediately [15, 16], as $\mathcal{S}(0, M, 10\,\text{ms}) < g_0$. A processes the empty event and immediately releases the tag $(0, M)$ [17, 18]. Consequently, B can now also process the empty event and release the tag [20]. This, in turn, allows A to acquire the tag $(10\,\text{ms}, 0)$ [19, 21].

A processes the message received from B and releases the tag [22]. It then tries to acquire the tag $(30\,\text{ms}, 0)$ from the incoming connection [23] to process the next triggering of its timer. As B has not released the tag yet, this schedules an empty event at $(20\,\text{ms}, M)$ in B. B tries to acquire the newly created tag [25], and since A has not released it yet, this schedules a new empty event in A [26]. A, in turn, immediately acquires the tag of the newly created event [27], which schedules another empty event with tag $(10\,\text{ms}, M)$ in B [28] and leads to a new empty event in A [29, 30]. Steps [27] to [30] are a repetition of steps [23] to [25]. Essentially, the acquire operation walks back both in time and on the path of the loop, subtracting 10 ms on each iteration.

A's acquire operation for tag $(10\,\text{ms}, M)$ succeeds immediately [31, 32], as B already released the tag $\mathcal{S}(10\,\text{ms}, M, 10\,\text{ms}) = (0, M)$. A processes the empty event, releases the tag $(10\,\text{ms}, M)$ [33, 34], and tries to acquire the next tag at $(20\,\text{ms}, M)$ [35]. B can now also process the empty event at tag $(10\,\text{ms}, M)$ and release the tag [36], which in turn allows A to acquire the tag $(20\,\text{ms}, M)$. This process (steps [33] to [37]) repeats until A acquires the tag $(30\,\text{ms}, 0)$ [43]. Essentially, the release messages circle back on the path taken by the acquire message.

Both acquire and release messages may circle in a loop for several iterations if the sum of delays on the connections within the loop is less than the interval in which new events are produced. The smaller the delay, the more iterations are required. While the proposed coordination mechanism still ensures correct execution, it can become inefficient if delays are not chosen carefully.

While it seems obvious from the sequence diagram that repeated iterations of messages are not necessary, deriving a coordination mechanism that works efficiently for cycles is challenging. This is because the enclaves are not aware that they are in a loop. In addition, the acquire operation only asks for permission to execute a certain tag, but it gives no guarantee that the sender will not send messages at an earlier tag. If A, for instance, has a physical action or receives messages from another enclave, then A needs to acquire tags from all its inputs. Thus, when A acquires the tag g from B, this does not imply that it has acquired g from all of its inputs. In fact, A may still receive earlier events from its other inputs.

One possible solution would be to make the enclaves that are in a cycle aware of the cycle. This could be done statically as part of the compilation or

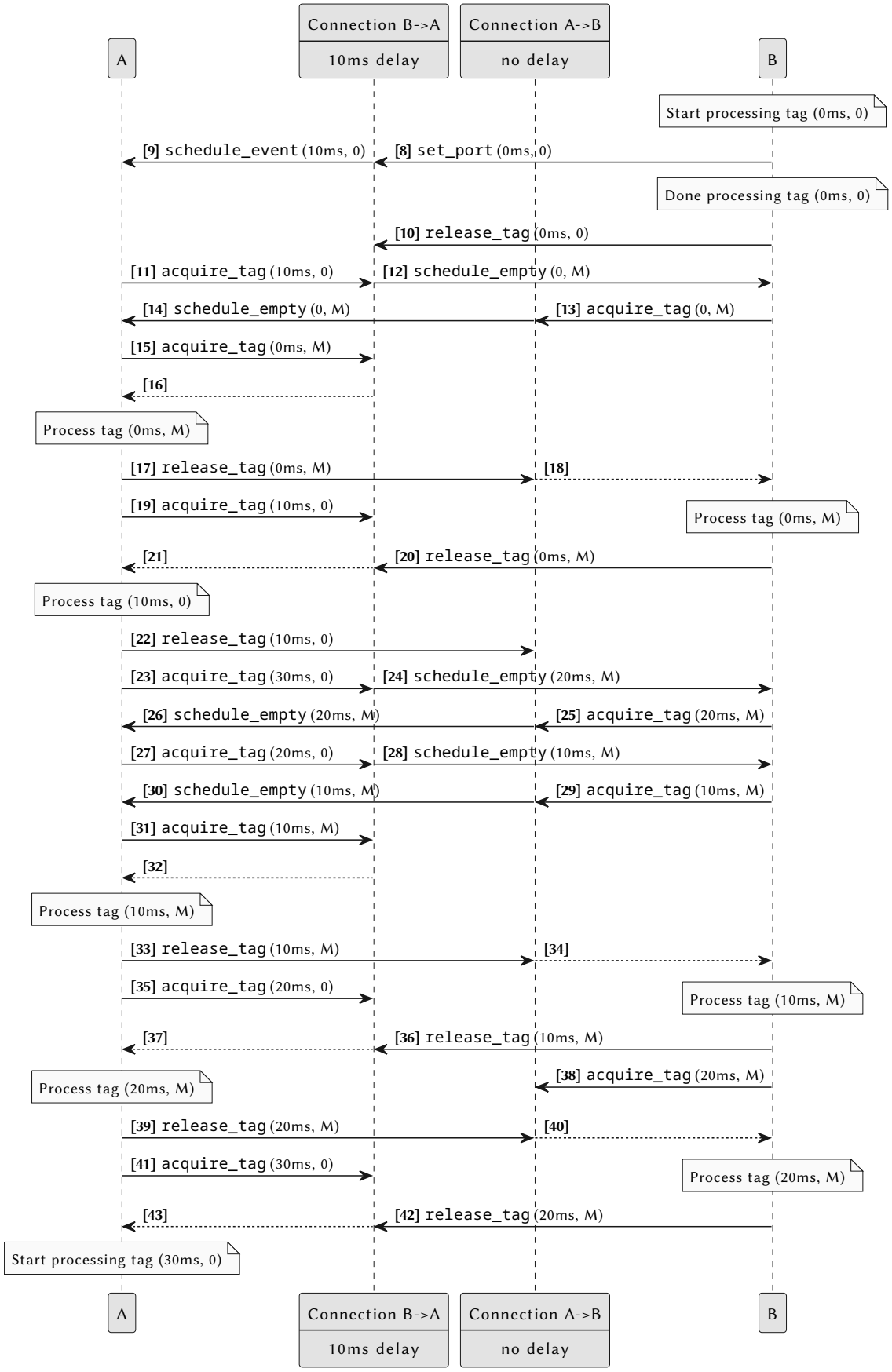


Figure 6.23: A sequence diagram that shows the communication between A and B for the program in Figure 6.21 with a 10 ms delay. Messages [1] to [7] are omitted because they are identical to the first messages in Figure 6.22.

dynamically using a discovery protocol. We could require that any enclave within the cycle that intends to process tag g first acquires g from all its other inputs that are not part of a cycle. Then the enclave could release all tags before g on its outgoing connection within the cycle and thereby guarantee that it would not produce any events before g. Finally, it would acquire g which succeeds as long as there is at least a microstep delay along the cycle. This approach, however, only works if each enclave is at most part of one cycle.

Another solution is presented by the centralized coordination used for federated execution in LF. This relies on the central RTI to decide when it is safe to process a certain tag. This also works reliably for federates with cyclic connections. However, this chapter has argued that a single centralized coordinator introduces a significant bottleneck and a single point of failure. The coordination mechanism discussed in this chapter highlights that similar results can be achieved with a peer-to-peer coordination mechanism. However, the RTI solution is more efficient in the presence of cycles. Therefore, a possible solution could combine both approaches. While acyclic subsets of an application may use peer-to-peer coordination, local RTIs could be introduced to coordinate within each cycle. In fact, one of the enclaves within the cycle could be elected to become the coordinator of this cycle.

6.6.2 Cycles without Delays

While cycles with delays are in principle supported by the discussed enclave coordination mechanism, although the protocol may be inefficient, cycles without any delay are currently not supported. Consider the example program in Figure 6.24. This program is a legal reactor program, as the APG that denotes the dependencies between reactions is acyclic. The topology graph, however, exposes a cycle between enclaves that is not broken up by any delays.

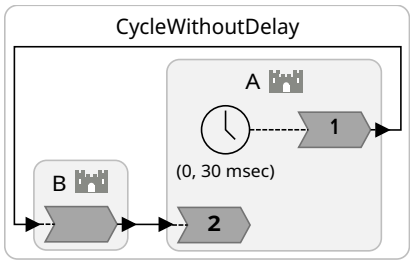


Figure 6.24: An example program with cyclic connections without delays between enclaves.

When this example program is executed using a single scheduler (without enclaves), the scheduler relies on the levels assigned to reactions based on the APG. For every triggering of the timer, it would first execute reaction 1 of A, then B's reaction, and finally reaction 2 of A. However, if we consider A and B as individual programs, then it is not obvious for the scheduler of A that it should first execute reaction 1, then wait for a message from B, and finally execute reaction 2.

Based on the coordination mechanism introduced in this chapter, A would try to acquire the startup tag from B, which in turn would acquire the same tag from A. The execution immediately deadlocks, as none of the enclaves can release the tag requested by the other. To resolve this, we would need a mechanism for representing levels between enclaves. One possible way would be to introduce another dimension to the tags to also account for levels. Then all connections would introduce at least a level delay. This additional dimension would only be used to order events within a tag. A reactor would

still consider two events with the same base tag but different levels to be logically simultaneous.

Another possibility for supporting loops without delays could be a more fine-grained coordination scheme. Instead of acquiring tags before starting to process an event and releasing tags only after all reactions at this tag have completed executing, we could acquire the inputs required at each level. And once all reactions that may write to an output are completed, we could release the current tag on this output. For such a scheme to work, the level assignment algorithm would also need to consider dependencies outside an enclave. Thus, either static assignment of levels at compile time or a dynamic discovery algorithm would be required.

6.6.3 Manual Partitioning

While this chapter illustrates how enclaves can be used to partition programs to improve parallelism or ensure that certain timing requirements are met, this approach fully relies on manual annotations. However, it is typically not obvious to the programmer that a program requires partitioning. For instance, in the car brake example, only experimentation and tests reveal that there may be deadline misses, but this is not apparent from the LF code or from the generated diagram. Currently, the LF tooling does not provide sufficient feedback or guidance to support programmers in deciding if programs need to be partitioned and which reactors should become enclaves.

One possible solution would be to make every reactor an enclave. In this case, each reactor would have its own scheduler and could operate autonomously, only coordinating with its direct dependencies. This would maximize the decoupling between components. In this case, reactors would only need to wait on their direct dependencies, and we could maximize the exploitable parallelism. However, this comes at the cost of significant overhead for coordinating the individual reactors. This overhead can be acceptable if reactions are computationally heavy and the gains in parallelism outweigh the overhead. Moreover, if certain properties of the application (e.g., deadlines) require decoupling, some additional overhead is often acceptable. However, this is not the case for all applications, and there is a trade-off between the benefits gained from decoupling components and the overhead this decoupling introduces. Using no enclaves or making all reactors enclaves are too extremes on a spectrum, and most often the optimal solution will lie in between.

In principle, the LF compiler could also be extended to automate partitioning. It could utilize heuristics and DSE techniques to identify solutions that are near optimal regarding certain constraints. Such a compiler extension could not only decide how to partition a program, but also reason about the resources assigned to each enclave. Since every enclave has its own scheduler and pool of worker threads, we also need to decide how many threads to assign to each enclave. If the target architecture is heterogeneous, this may also include mapping threads to particular processing resources. Chapter 7 introduces a DSE framework that could be utilized for exploring such a solution.

6.7 Conclusion

This chapter introduced the concept of enclaves and discussed how enclaves are implemented in the C++ target of Lingua Franca. Based on selected

examples, the discussion showed that enclaves can effectively decouple reactors within a program to better exploit parallelism or ensure that certain deadlines are met. In addition, enclaves allow for implementing various patterns in LF that mimic the semantics of other MoCs like Hewitt actors and the LET paradigm.

While enclaves are conceptually closely related to federates, the current C++ implementation explores a coordination mechanism that significantly differs from the centralized and decentralized coordination schemes that are currently available for federated execution. The newly developed coordination mechanism highlight that reactors can be coordinated deterministically without relying on a central coordinator (RTI) or requiring detailed WCET estimations. However, this approach currently has severe limitations, in particular in conjunction with cycles in the reactor topology.

Likely, future work will aim to combine the different coordination mechanisms into a more unified approach. The abstractions introduced in this chapter provide reactors with the means to communicate through an arbitrary medium using arbitrary protocols. Thus, the coordination mechanism introduced in this chapter could be combined with the centralized and decentralized mechanisms used for federations. In such an approach, we could select the best protocol for each individual connection. While sequential reactors could use the mechanism proposed in this chapter, reactors in a cycle could consult a (local) RTI. And if a connection should favor availability over consistency, then this could be indicated by a user annotation, and we could use the decentralized coordination scheme based on PTIDES.

Future work will also need to consider better mechanisms for analyzing LF programs and prompting users with useful feedback that allows them to make educated decisions on how to partition the program. Additionally, we could develop tools that automatically explore the design space of possible solutions and utilize heuristics to identify solutions that are near optimal regarding certain constraints. The next chapter introduces a DSE framework that has been used successfully for optimizing dataflow applications and that is flexible enough to also allow reasoning about reactor programs. This framework could become the foundation for integrating DSE techniques into the LF toolflow and researching strategies for partitioning LF programs automatically.

Design Space Exploration with Mocasin

7

Scalability, as discussed in Section 1.2.4, not only refers to the ability to manage software complexity and generate efficient realizations but also requires managing hardware complexity, which is increasing rapidly. With the rise of the multicore era, MPSoCs have become ubiquitous, even in the embedded domain. Due to the end of Moore's law,[1] hardware designers meet the ever-increasing demands for computational power by specializing and optimizing hardware for specific uses. Heterogeneous MPSoCs integrate a multitude of different processing elements and accelerators, as well as different memory architectures.

Early examples of heterogeneous architectures include the Odroid-XU4 shown in Figure 7.1, which is based on a Samsung Exynos 5422 MPSoC that uses ARM's big.LITTLE architecture.[2] It has two types of cores: the little cores (Cortex A7) are more energy efficient, and the big cores (Cortex A15) are more performant. Other examples include the Tomahawk architecture,[3] Kalray's MPPA Coolidge with 80 processing elements,[4] the SpiNNaker 2[5] for neuromorphic computing, or the Xilinx Zynq UltraScale+ architecture,[6] which integrates a field-programmable gate array (FPGA) with a heterogeneous MPSoC.

Programming such MPSoCs becomes increasingly challenging. Not only do we need to manage the challenges of concurrency to exploit parallelism, but we also need to carefully allocate chunks of computation to the different computation resources. The heterogeneous nature of modern MPSoCs opens up a huge design space. Moreover, different computing resources, such as conventional processors, accelerators, DSPs, graphics processing units (GPUs), and FPGAs, need to be programmed differently. Due to the complexity of the underlying hardware, programmers spend a significant amount of time managing the complexity and less time implementing algorithms. Jeronimo Castrillon has identified this problem as the *software productivity gap*.[7]

There are a range of design space exploration (DSE) tools that aim to close this gap.[8] These tools, at their core, utilize the hourglass model (cf. Section 1.1 and Figure 7.2). Programmers design applications on top of a MoC and DSE tools and compilers are responsible for synthesizing concrete realizations on a specific target architecture. This process assigns concrete computation and communication resources in hardware to the software elements of the application. We refer to this process as *mapping*.

While existing DSE tools are typically based on dataflow or process network MoCs, a similar approach can be taken for compiling Lingua Franca programs on heterogeneous architectures. In the context of LF, we can also add another dimension to the DSE problem. The exploration can also aim to identify the best partitioning of a given LF program and then assign hardware resources to the partitions.

This chapter provides a general introduction to existing DSE toolflows and introduces the Mocasin framework, which is a core contribution of this thesis. In contrast to existing DSE toolflows, which are primarily designed to support application designers, Mocasin is designed as an open research framework for exploring and developing new approaches to DSE itself. This chapter demonstrates Mocasin's ability to model and analyze new DSE strategies in a case study, that uses Mocasin to prototype and evaluate a hybrid mapping strategy for an LTE base station.

1: Moore 1965, *Cramming More Components Onto Integrated Circuits*; Sutter 2005, *The Free Lunch Is Over: a Fundamental Turn Toward Concurrency in Software*.

2: Roy and Bommakanti 2017, *Odroid-XU4 User Manual*.

3: Haas et al. 2017, *A Heterogeneous SDR MPSoC in 28 Nm CMOS for Low-Latency Wireless Applications*; Castrillon, Lieber, et al. 2018, *A Hardware/software Stack for Heterogeneous Systems*.

4: Kalray Inc 2023, *MPPA DPU Architecture*.

5: Mayr, Hoeppner, and Furber 2019, *SpiNNaker 2: A 10 Million Core Processor System for Brain Simulation and Machine Learning*.

6: Advanced Micro Devices, Inc. 2023, *Zynq UltraScale+ MPSoC*.

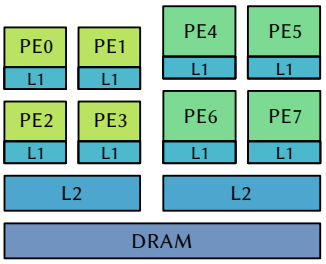


Figure 7.1: Architecture of the Odroid-XU4 with four little cores (Cortex A7) and four big cores (Cortex A15).

7: Castrillon, Sheng, and Leupers 2011, *Trends in Embedded Software Synthesis*; Castrillon and Leupers 2014, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*.

8: Castrillon, Desnos, et al. 2023, *Dataflow Models of Computation for Programming Heterogeneous Multicores*.

Mocasin focuses primarily on dataflow and process network MoCs. However, in principle, the flexible and modular architecture of Mocasin also allows for integrating the reactor model. While such an integration remains for future work, this chapter lays the foundation for DSE in Lingua Franca and sketches possible approaches for integrating with Mocasin.

Many of the figures, results, and discussions presented in this chapter have been published before in Menard, Goens, Hempel, et al. 2021, *Mocasin—Rapid Prototyping of Rapid Prototyping Tools: A Framework for Exploring New Approaches in Mapping Software to Heterogeneous Multi-cores* and Castrillon, Desnos, et al. 2023, *Dataflow Models of Computation for Programming Heterogeneous Multicores*. Similar to Lingua Franca, Mocasin is a highly collaborative effort with several contributors. While the core ideas behind Mocasin and the main architecture are contributions of this thesis, many of the algorithms and mapping strategies implemented in Mocasin are either based on prior work or were contributed by coauthors.

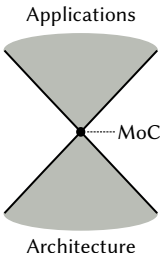


Figure 7.2: The hourglass model. (Repeated from Figure 1.1)

7.1 Design Space Exploration

Most existing DSE tools specialize in dataflow or process networks, in particular, SDF and KPN. Mapping an SDF or KPN application requires assigning actors to particular processing resources in the target architecture and channels to the available communication resources. Since the KPN and SDF models do not assume shared memory, we can freely choose the type of processing resource and also distribute actors across a network of computers. Since the models are deterministic, we further know that, when implemented correctly, the concrete mapping will not influence the behavior of the application.

Even for seemingly small applications and target architectures, the space of possible mappings can be vast. Commonly, the mapping space is also shaped irregularly with many local minima and maxima.[9] DSE tools automate the process of navigating this mapping space and utilize heuristics to determine near-optimal mappings while respecting a given set of optimization criteria and constraints. This section gives a brief introduction to DSE and provides an overview of existing tools. The interested reader may find a more in-depth discussion in Castrillon, Desnos, et al. 2023, *Dataflow Models of Computation for Programming Heterogeneous Multicores*.

9: Goens 2021, *Improving Model-Based Software Synthesis: A Focus on Mathematical Structures*.

Figure 7.3 shows the typical flow of DSE tools. Starting from a description of the application and the target platform, DSE tools apply a mapping algorithm to identify good mappings. While we can statically analyze the behavior of SDF applications, the toolflow requires additional information about the execution of KPN. This is typically given in the form of traces that are recorded on the host machine and that represent one possible execution of the application. Common mapping algorithms are meta-heuristics that walk the design space. To guide this search, a simulator or performance predictor estimates the quality of a mapping regarding the optimization criteria (e.g., makespan or energy consumption). Once a mapping is selected, the tool can synthesize software or hardware to implement the actors and the channels of the input program on the selected hardware components.

The literature describes a wide range of DSE tools. The PREESM framework, for example, is a specialized toolflow for πSDF applications.[10] It is not limited to automatically deriving mappings and schedules for given applications on a given platform; it can also be used for hardware/software co-design, where

10: Pelcat et al. 2014, *PREESM: A Dataflow-based Rapid Prototyping Framework for Simplifying Multicore DSP Programming*.

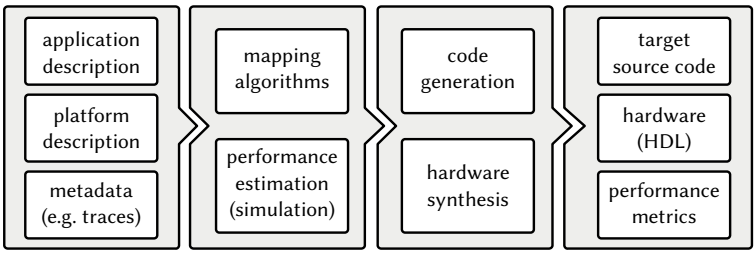


Figure 7.3: Generalized DSE toolflow for mapping applications to heterogeneous multi-core architectures.

the performance predictions made by the DSE tool guide the development of the hardware platform to meet the requirements of a specific application. There are also hybrid DSE flows that narrow the design space statically but only select concrete mappings during execution of the application. PREESM can be used with the SPIDER runtime to support dynamic reconfiguration of πSDF applications.[11]

There are also several DSE frameworks based on the KPN MoC. Sesame, for instance, is a framework with a strong focus on DSE and simulation at multiple levels of abstraction.[12] Sesame is part of the Daedalus tool[13] and can be used in combination with ESPAM[14] to directly synthesize optimized hardware from dataflow applications. A similar approach is also taken by SystemCODesigner.[15]

MPSoC Application Programming Studio (MAPS) is another KPN-based framework. It provides a C extension (called CPN) for describing applications and comes with a rich set of mapping algorithms and analysis tools, including a high-level trace-based simulator.[16] A similar simulator is also used in the Turnus DSE framework,[17] which simulates traces of dynamic dataflow applications written in the CAL actor language (CAL).[18]

The DOL and DAL frameworks for KPN applications include analytical performance estimation alongside a system-level SystemC simulator for more general platforms.[19] DAL supports an extended KPN model, including scenario state machines and additional control channels. The analytical model uses real-time calculus and is restricted in the types of resources and schedulers it can handle.

All the discussed DSE flows are specializations of the generalized flow in Figure 7.3. They focus only on a particular MoC, a particular application domain, or a particular type of target hardware. Such specialization can simplify the overall architecture and help narrow the exploration by only focusing on a particular domain, which improves the overall user experience. However, while specialization allows for focusing on specific problems, it also limits the possibilities for innovation. In research, we need to prototype and analyze new approaches, which is often not possible within the constraints of existing tools. Moreover, comparing different approaches implemented in different tools with different constraints is challenging.[20]

7.2 Mocasin

This section introduces Mocasin, an open-source rapid prototyping framework for exploring new DSE approaches for mapping software to heterogeneous multi-cores.[21] In contrast to existing tools, Mocasin is specifically designed to support researchers and developers working on DSE flows. It is not intended as a workflow for end-users. Mocasin is a complementary research tool that allows for exploring potential improvements to existing workflows and prototyping customized workflows for new use cases.

11: Heulot, Pelcat, et al. 2014, *SPIDER: A Synchronous Parameterized and Interfaced Dataflow-based RTOS for Multicore DSPs*.

12: Pimentel, Cagkan. Erbas, and Polstra 2006, *A Systematic Approach To Exploring Embedded System Architectures At Multiple Abstraction Levels*.

13: Nikolov, Thompson, et al. 2008, *Daedalus: Toward Composable Multimedia MP-SoC Design*.

14: Nikolov, Stefanov, and Deprettere 2008, *Systematic and Automated Multiprocessor System Design, Programming, and Implementation*.

15: Keinert et al. 2009, *Systemcodesigner—An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications*.

16: Leupers and Castrillon 2010, *MPSoC Programming Using the MAPS Compiler*; Castrillon and Leupers 2014, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*.

17: Casale-Brunet et al. 2013, *Turnus: A unified dataflow design space exploration framework for heterogeneous parallel systems*.

18: Eker and Janneck 2003, *CAL Language Report: Specification of the CAL Actor Language*.

19: Thiele et al. 2007, *Mapping Applications to Tiled Multiprocessor Embedded Systems*; Schor et al. 2012, *Scenario-Based Design Flow for Mapping Streaming Applications onto on-Chip Many-Core Systems*.

20: Goens, Khasanov, Castrillon, et al. 2016, *Why Comparing System-Level MP-SoC Mapping Approaches is Difficult: A Case Study*.

21: Menard, Goens, Teweleitt, et al. 2023, *Mocasin*.

7.2.1 Overview

Mocasin is designed from the ground up for increased flexibility and interoperability. Instead of specializing the flow in Figure 7.3 for a specific use case, Mocasin provides modular and general implementations of individual components. This includes data structures for various MoCs, abstractions for hardware platforms, several predefined mapping algorithms, a high-level simulator for performance estimation and evaluation of mapping quality, as well as several convenience tools (e.g., for visualization). All of Mocasin's components are configurable and exchangeable. This modular approach makes Mocasin an ideal toolbox for building customized flows, prototyping new mapping strategies and data structures, and evaluating the effects of such new approaches.

Mocasin combines the various approaches found across existing DSE tools to create a generic and flexible toolbox independent of specific use cases. However, Mocasin is not a replacement for these tools. It is a complementary framework that is designed from the ground up for interoperability. The main design goal is to facilitate the research of new approaches, the prototyping of improvements for existing tools, and the comparison of approaches across tools.

The complete architecture of Mocasin is shown in Figure 7.4. In essence, Mocasin is a toolbox that provides several modules, where each module focuses on a specific problem and may interact with other modules. Mocasin provides various *tasks*, each of which offers a unique functionality. Tasks can be considered a concrete instance of a tool flow that is composed of selected Mocasin components. The `visualize`task, for instance, opens a graphical user interface (GUI) that visualizes a platform as well as a spatial mapping of a given application on this platform. More elaborate tasks include `simulate` and `generate_mapping`which respectively execute a high-level simulation to estimate the performance of a given mapping or use a configurable mapping algorithm to find mappings.

7.2.2 Data Structures

Mocasin provides internal data structures for representing applications, platforms, mappings, and additional information about the runtime behavior, such as pre-recorded traces. To account for interoperability with other tools, these data structures are designed to be abstract and generic without making too many assumptions about a precise use case.

Each data structure is defined with a common base class that needs to be implemented by any object representing an application, platform, mapping, or trace. Thus, Mocasin does not impose restrictions on how such objects are created. While Mocasin provides a few standard methods, including file readers for various formats, a platform designer, and mapping generators, arbitrary new methods can be added via plugins.

Application

Applications are modeled as directed graphs, where nodes denote computation and edges represent data dependencies. Depending on the particular MoC, nodes or edges may be annotated with additional information, e.g., fixed token sizes for data channels or firing rates of nodes. This simple graph description matches the abstractions used in common dataflow MoCs including task graphs, SDF, KPN, Hewitt actors, and also reactors. Note that this

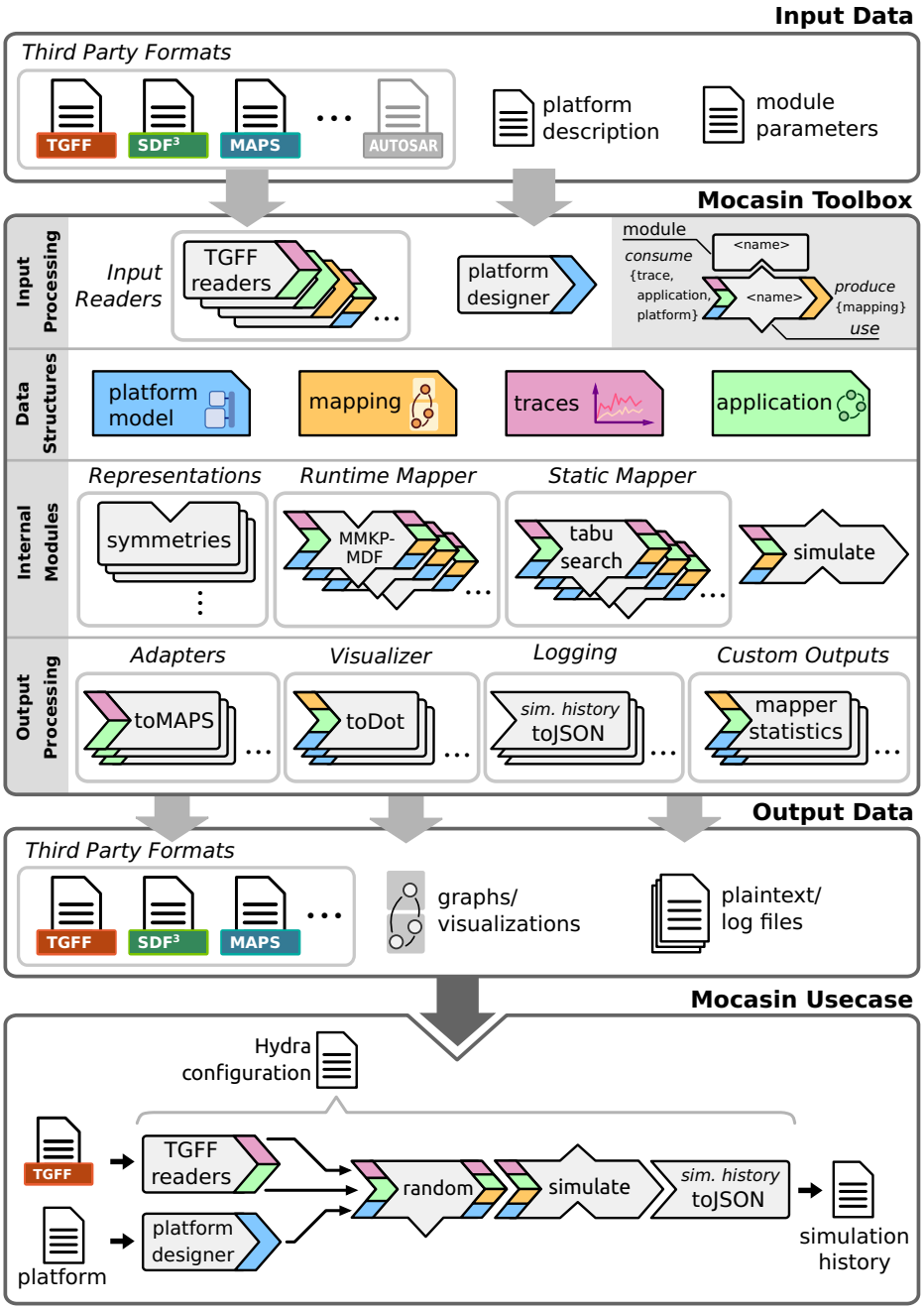


Figure 7.4: The Mocasin architecture. This figure was designed by Gerald Hempel.

application model only describes the topology of the application and does not provide information on its behavior.

Trace

A trace in Mocasin is a data structure that is complementary to an application. It describes one possible behavior of the application and represents a sample execution. Traces are the foundation for running simulations and for obtaining additional information as required for some mapping strategies. MoCs with strict firing rules like SDF and task graphs provide sufficient information to precisely specify the application's execution patterns. For these MoCs, traces can be automatically generated and simply encode the

firing rules. For more permissive MoC like KPN or Hewitt actors, however, the behavior is not defined statically. Thus, traces need to be recorded while executing a real implementation of the application. KPN-based frameworks like MAPS, Turnus or DOL/DAL can be used to instrument applications and obtain execution traces. But also general-purpose tracing frameworks can also be used to manually instrument an application to record relevant events and obtain traces.

An application trace in Mocasin is a sequence of segments, where each segment represents an action that a node in the application graph performs. A segment can denote a consume operation that reads some tokens from an incoming data channel, a produce operation on an outgoing data channel, or a computation lasting for a number amount of processor clock cycles. A special termination segment marks the end of the trace. To model computation on various platforms and types of processing elements, the trace can define different computation costs for different types of processing elements.

Platform

Mocasin models platforms as a set of *processing elements* and *communication primitives*. A processing element can represent any component capable of performing computation, like general-purpose processors, DSPs, or accelerators. Processing elements are characterized by a frequency and, if this is applicable, by an estimated cost in clock cycles required for a context switch on this processing element. Each processing element is also associated with a scheduler, which manages the workload executing on one or more processing elements according to a selected policy. Processing elements can also be annotated with a simple power model that describes the static and dynamic power consumption and that can be used for estimating costs in terms of energy consumption.

Communication primitives abstractly describe a mechanism for communicating data between processing elements. They are based on the primitives described by Castrillon and Leupers[22] but were extended for improved flexibility and accuracy.[23] Each communication primitive defines a set of source and sink processing elements that can use this primitive. For any pair of sink and source processing elements, multiple communication primitives may be defined if multiple mechanisms for exchanging data between these processing elements exist on the real platform. Each communication primitive defines two lists of *communication phases*—one for the producing side and one for the consuming side.[24] Each phase represents one step in the communication processes and defines a list of *communication resources* that it requires. Resources represent the actual hardware used to move data along a certain path in the platform (e.g., buses, links, caches, scratchpad memories, DRAMs or DMAs. Each resource is defined by its read and write latency and total throughput.

22: Castrillon and Leupers 2014, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*.

23: Menard, Goens, and Castrillon 2016, *High-Level NoC Model for MPSoC Compilers*.

24: Odendahl et al. 2013, *Split-cost communication model for improved MPSoC application mapping*.

In summary, each communication primitive provides step-by-step instructions on how a pair of processing elements can exchange data and how the precise communication costs for each step can be calculated. This mechanism provides a lot of flexibility and can be used to model a wide range of architectures, including bus-based, clustered, and network-on-chip (NoC) based architectures,[23] as well as distributed systems.

Figure 7.5 shows an example platform model that represents the Odroid-XU4. Square nodes represent the processing elements, and oval nodes denote communication primitives. An edge from a processing element to a communication primitive indicates that the processing element can write a data

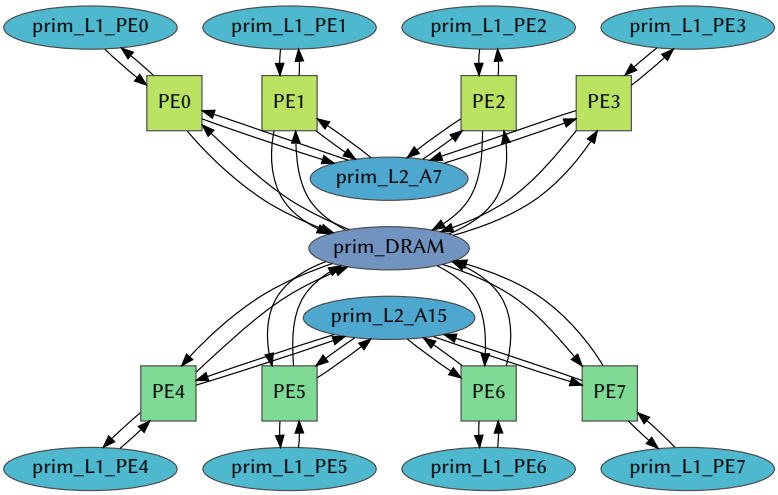


Figure 7.5: A visualization of the platform graph used by Mocasin internally to represent the Odroid-XU4 architecture.

token using this communication primitive. An edge from a communication primitive to a processing element indicates that the processing element can read a data token using this communication primitive.

The Odroid platform has two clusters of processing elements: one consisting of four ARM Cortex-A7 cores (little) and one consisting of four ARM Cortex-A15 cores (big). Each core has its own L1 cache, and each cluster shares an L2 cache. Both clusters have access to the DRAM via a shared bus. There is a communication primitive for each memory in the system. For instance, all processing elements can communicate with each other by writing to the DRAM and reading from it. Within each cluster of little and big processing elements, the L2 caches can be used. And for communication between different processes on the same core, the L1 caches can be used.

Mocasin's platform model effectively abstracts from the precise platform topology. It focuses on modeling computation costs and data communication mechanisms. This abstract view is compatible with many platform description formats as they are found in related tools as well as in industry standards like IEEE 2804-2019 (SHIM)[25] and AUTOSAR.

25: IEEE Computer Society 2019, *IEEE Standard for Software-Hardware Interface for Multi-Many-Core*.

Mapping

A mapping assigns the nodes and edges of a given application graph to processing elements and communication primitives on a target platform. This is implemented as a simple dictionary. Each of the assignments may be annotated with additional information, like a process priority or a maximum channel capacity. Mappings can also be provided as a sequence over time to implement a fixed schedule, as is commonly done for SDF applications. In addition to this simple dictionary view of a mapping, the representations implemented in Mocasin (cf. Section 7.2.5) provide more sophisticated views that can be utilized by various algorithms.

Readers

To obtain the data structures described above, Mocasin provides modular readers that create the internal data structures by reading from input files. The abstract models used by Mocasin enable conversion from a wide range of existing formats and tools. To illustrate this flexibility, Mocasin currently provides readers for SDF applications in the SDF for Free (SDF^3) format,[26] task graphs in the Task Graphs for Free (TGFF) format,[27] and KPNs in the

26: Stuijk, Geilen, and Basten 2006, *SDF^3: SDF For Free*.

27: Dick, Rhodes, and Wolf 1998, *TGFF: Task Graphs for Free*.

```
# Select processors from library
little_processor = Processor.get("LITTLE", type_="ARM_CORTEX_A7")
big_processor = Processor.get("BIG", type_="ARM_CORTEX_A15")
# Add two cluster of processors
designer.addPeClusterForProcessor("cluster_a7", little_processor, 4)
designer.addPeClusterForProcessor("cluster_a15", big_processor, 4)
# Add L1 caches to each processor
designer.addCacheForPEs("cluster_a7", name='L1', <params>)
designer.addCacheForPEs("cluster_a15", name='L1', <params>)
# Add L2 caches to each cluster
designer.addCommunicationResource("L2_A7", ["cluster_a7"])
designer.addCommunicationResource("L2_A15", ["cluster_a15"])
# Add a RAM accessible by all PEs
designer.addCommunicationResource("RAM", ["cluster_a7", "cluster_a15"])
```

Listing 7.1: Description of the Odroid-XU4 board in Python using Mocasin's platform designer API.

MAPS format. More readers for other tools and formats can be easily added by extending Mocasin directly or by providing a plugin implementing the reader.

Supported MAPS formats comprise a description format for KPN-based applications, a platform description format that is close to the SHIM standard, an internal execution trace exchange format, and a mapping exchange format. The Mocasin readers automatically convert all four MAPS file formats to the internal data structures.

The SDF^3 and TGFF formats describe applications based on the SDF and task graph MoCs, respectively. Since the semantics of these MoCs define strict firing rules, SDF^3 and TGFF files provide sufficient information for generating both Mocasin's application and trace data structures. Additionally, SDF^3 provides platform descriptions and mapping formats. However, importing these descriptions is not yet supported.

7.2.3 Platform Designer

A central enabler in researching compilation methods for complex architectures is system modeling. Depending on the level of abstraction and fidelity required, this can be an extremely complex endeavor or a fairly simple matter. During our work on and with Mocasin we experienced the implementation of new platforms as an elaborate and time-consuming task.

The communication primitive abstraction of Mocasin's platform model is useful for estimating communication delays, but it is not a straightforward method for describing the topology of such architectures. Therefore, Mocasin provides an API called platform designer that allows for describing the topology graph. Based on this topology description, the platform designer automatically creates a platform model and generates all communication primitives.

The code excerpt in Listing 7.1 illustrates how the platform designer API can be used to describe the Odroid-XU4 platform. The code excerpt describes precisely the topology. It defines the two clusters of little and big cores and adds caches as well as the DRAM as communication resources. Note that the example omits the precise parameters of hardware components like frequency, throughput, and latency for brevity. From this description, the platform designer automatically generates the platform model using communication primitives, as shown in Figure 7.5.

The platform designer is also capable of describing NoC-based architectures. Elements can simply be connected by providing parameters describing the NoC characteristics and an adjacency matrix. Mocasin also provides

a set of predefined platforms utilizing the platform designer to model the Odroid-XU4 as described above, as well as configurable platform models that represent common patterns such as mesh-based NoC topologies or bus-based hierarchical architectures.

7.2.4 Simulator

The simulation module is a key component of Mocasin. It implements a high-level simulator capable of estimating the performance of given applications (consisting of an application graph, mappings, and traces) running on a given platform. This not only enables rapid performance estimation, but it is also the key enabler for evaluating the characteristics of various MoCs, mapping algorithms, and representations within Mocasin. While the simulator aims to provide accurate results, it neither models the hardware nor the software running on top precisely. Instead, it uses abstractions that capture the essence of the hardware characteristics and the application behavior. Related tools implement similar high-level simulators for performance estimation. However, Mocasin's simulator is designed for increased flexibility, supporting various dataflow MoCs and also allowing other components to interact with the simulation, as the case study in Section 7.3 illustrates.

Mocasin's simulator is based on the SimPy discrete-event simulation framework.[28] The basic simulator structure is designed to be independent of the concrete MoC semantics, application behavior, and hardware characteristics. Essentially, an application is modeled as a set of concurrent processes interacting with each other. The precise semantics of this interaction can be adjusted to implement concrete MoCs.

28: SimPy 2023, *SimPy: Discret Event Simulation for Python*.

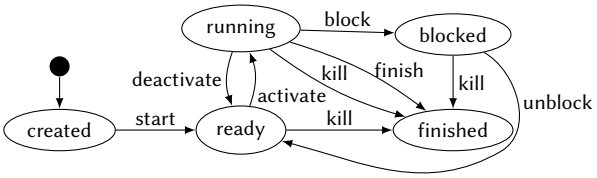


Figure 7.6: The basic process model simulated in Mocasin.

The execution of a process in the simulator is modeled abstractly as the well-known finite state machine shown in Figure 7.6, which is based on the classical model for POSIX threads. The execution of all processes in the system is controlled by a set of schedulers, each of which controls of one or more processing elements. Currently, Mocasin implements the FIFO and round-robin scheduling algorithms. Other algorithms can be added via plugins.

The process model shown in Figure 7.6 is an abstraction that separates the basic processing mechanisms and scheduling algorithms implemented in the simulator from the concrete MoC semantics and application behavior. The precise semantics are implemented on top of this abstraction. This is analogous to a real-world runtime that implements MoC semantics through an abstraction layer on top of a thread model. In Mocasin, concrete MoC semantics can be implemented by specializing the process class provided by Mocasin overriding its `workload`method.

For instance, Mocasin implements the KPN MoC. The `workload`function of a KPN process implements the behavior of one node in the KPN application graph by replaying the trace provided for this node. The trace provides step-by-step instructions as to what the process needs to do. For instance, the trace could start with a consume segment reading a certain number of data tokens from a channel, followed by a computation segment, and finally a produce segment writing a certain number of tokens to a channel.

The KPN implementation models the state of each FIFO channel that connects KPN nodes. Note that this state only entails the number of data tokens that are stored in the buffer and does not describe concrete data. For each read operation, KPN processes check the state of the corresponding input channel. If a sufficient number of tokens are available, the process retrieves the tokens from the channel. The delay imposed by the consume operation is calculated based on the concrete communication primitive selected by the mapping and specified by the platform model. If not enough tokens are available, the process blocks and thus waits until a sufficient number of tokens become available.

A compute segment simply delays the execution by a certain time. The precise time is calculated based on the cycle count provided in the trace for the given processing element that the process is mapped to. Writing tokens is implemented analogously to reading. If there is not enough free space in the modeled FIFO buffer, the process blocks. Otherwise, it continues and accounts for any communication delays. If another process is waiting to read tokens from this channel, it will be automatically unblocked.

While the above description focuses on the KPN MoC, the generic simulation infrastructure can be utilized to model arbitrary MoCs and runtime strategies. This includes the modeling of dynamic workloads, such as a runtime scheduler that assigns incoming tasks to a number of worker threads (cf. Section 7.3).

The simulator can also produce a history in JSON format, which provides detailed information about the simulated execution and is a good basis for further analysis. The JSON history can be visualized with the Catapult Trace Viewer[29] as shown in the example in Figure 7.7.

29: Google 2023, *Catapult*.

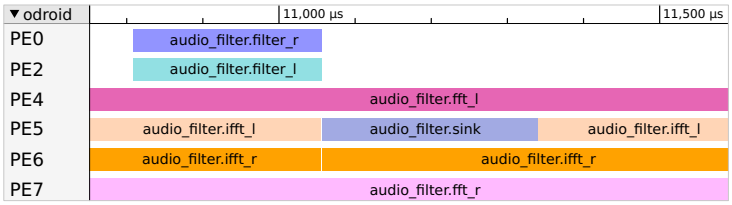


Figure 7.7: Visualization of the simulated execution of an audio filter application on the Odroid-XU4.

7.2.5 Representations

Representations are a unique idea behind the modular design in Mocasin and are concerned with mathematical encodings of a mapping.[30] The most common way to represent this in algorithms is what we call the `SimpleVector` representation. A mapping m is specified as a vector:

$$m = (p_1, \dots, p_k, c_1, \dots, c_l)$$

where the p_i are processing elements for each of the $i = 1, \dots, k$ computational tasks (or actors or processes) and the c_j are communication primitives for the data. Many algorithms consider communication implicitly or just ignore it, removing the c_j from the representation. However, other representations are possible, like an embedding to real vectors that captures a distance metric between processors[31] or the symmetries of the architecture.[32]

The choice of representation directly influences how the mapping space is shaped. Ideally, a good representation creates a mapping space that is shaped so that the global optimum can be found (more) easily.

30: Goens, Menard, and Castrillon 2018, *On the Representation of Mappings to Multicores*; Goens 2021, *Improving Model-Based Software Synthesis: A Focus on Mathematical Structures*.

31: Thompson and Pimentel 2013, *Exploiting Domain Knowledge in System-Level Mpsoc Design Space Exploration*; Goens, Menard, and Castrillon 2018, *On the Representation of Mappings to Multicores*.

32: Goens, Siccha, and Castrillon 2017, *Symmetry in Software Synthesis*; Schwarzer et al. 2018, *Symmetry-Eliminating Design Space Exploration for Hybrid Application Mapping on Many-Core Architectures*.

7.2.6 Mappers

Mappers in Mocasin are responsible for generating mappings for a given application (or multiple applications) on a given target platform. Mocasin defines a common mapper interface. This allows for integrating and testing new mapping algorithms simply by providing another implementation of the interface. The mappers can automatically leverage the various abstractions provided by Mocasin and utilize different representations. A simulation manager abstracts the process of evaluating a series of mappings to obtain performance estimations. This enables leveraging the structure of mappings when searching the design space, e.g., by getting a symmetry-aware cache[33] for free.

In general, a wide range of different algorithms can be used for generating mappings.[34] Mocasin implements several heuristics and meta-heuristics. Heuristics utilize domain knowledge and the information provided by Mocasin's internal data structures to derive a mapping. For instance, Mocasin provides a simplistic default mapper that maps everything to the first available processing element and communication primitive. There is also a random mapper, which assigns all processing elements and communication primitives randomly. Finally, the static fair mapper follows the basic design principle of the Linux Completely Fair Scheduler (CFS) scheduler[35] and implements a load balancing heuristic.

Meta-heuristics explore the design space of mappings by evaluating multiple candidates and refining them through the search. The implemented meta-heuristics range from a simple random walk to more sophisticated genetic algorithms. The genetic algorithms are implemented with the DEAP framework[36] and follow the general approach used in Sesame.[37] Mocasin also implements a tabu search mapping algorithm and a simulated annealing mapper based on prior work.[38]

Mocasin also supports scheduling, where the resources are reassigned at certain intervals during the execution. There are, for instance, a knapsack-based algorithm,[39] and one based on a Lagrangian relaxation method.[40] When comparing runtime scheduling with static mapping strategies (e.g., genetic algorithms), we can implement hybrid approaches and generate spatio-temporal mappings, like in TETRiS.[41]

Mappers with objectives apart from makespan or energy consumption are also supported by Mocasin. For instance, there is a bio-inspired design centering algorithm that searches for robust mappings.[42]

7.2.7 Configuration

All tasks provided by Mocasin can be configured via yaml files and command-line parameters. Mocasin uses Hydra for managing those configurations, which is a key enabler for its flexibility.[43] Hydra supports the dynamic composition of configurations from various sources, which allows users to combine external and internal modules to form flows tailored for specific use cases.

The use case depicted in Figure 7.4, for instance, reads the application graph and traces from a TGFF file, creates a platform model of the Odroid XU4 leveraging the platform designer, generates a random mapping, and simulates the application executing accordingly on the platform. This flow is executed by the following command:

```
mocasin simulate graph=tgff_reader trace=tgff_reader \
               platform=designer_odroid mapper=random
```

33: Goens, Siccha, and Castrillon 2017, *Symmetry in Software Synthesis*.

34: Singh et al. 2013, *Mapping on Multi/Many-Core Systems: Survey of Current and Emerging Trends*.

35: Molnar 2023, *Design of the CFS scheduler*.

36: Fortin et al. 2012, *DEAP: Evolutionary Algorithms Made Easy*.

37: Cagkan Erbas, Cerav-Erbas, and Pimentel 2006, *Multiobjective Optimization and Evolutionary Algorithms for the Application Mapping Problem in Multiprocessor System-On-Chip Design*; Quan and Pimentel 2014, *Towards Exploring Vast MPSoC Mapping Design Spaces Using a Bias-Elitist Evolutionary Approach*; Goens, Khasanov, Castrillon, et al. 2016, *Why Comparing System-Level MPSoC Mapping Approaches is Difficult: A Case Study*.

38: Manolache, Eles, and Peng 2008, *Task Mapping and Priority Assignment for Soft Real-Time Applications Under Deadline Miss Ratio Constraints*; Orsila et al. 2007, *Automated Memory-Aware Application Distribution for Multi-Processor System-On-Chips*.

39: Khasanov and Castrillon 2020, *Energy-efficient Runtime Resource Management for Adaptable Multi-application Mapping*.

40: Wildermann, Weichslgartner, and Teich 2015, *Design Methodology and Run-Time Management for Predictable Many-Core Systems*.

41: Goens, Khasanov, Hähnel, et al. 2017, *TETRiS: a Multi-Application Run-Time System for Predictable Execution of Static Mappings*.

42: Hempel et al. 2017, *Robust Mapping of Process Networks to Many-Core Systems Using Bio-Inspired Design Centering*.

43: Yadan 2019, *Hydra—A Framework for Elegantly Configuring Complex Applications*.

Each of the configuration keys can be adjusted as needed. For instance, the flow could also run the static CFS mapping heuristic by specifying `mapper=static_cfs` or read the application from an SDF³ file by specifying `graph=sdf3_reader`. Note that the selectable options are not limited to the modules provided by Mocasin. Leveraging hydra's plugin mechanism, external modules can be easily defined and included in the configuration. Also note that users can create customized configurations for their use case to avoid specifying all parameters as command-line arguments.

7.3 Case Study: Simulating a Hybrid Mapping Strategy for an LTE Base Station

This section presents a case study that illustrates how Mocasin and the tools that it provides can be leveraged to quickly prototype DSE flows for new use cases. Concretely, this section investigates an long-term evolution (LTE) telecommunications application. We create a toolflow for generating and evaluating hybrid mapping strategies in this domain.

One of the main challenges in current and upcoming telecommunications standards (5G and beyond) is the increased dynamicity of the workloads. When designing base stations, we need to consider that the computational load and constraints, like deadlines, may vary considerably depending on the number and types of data packets received. The hardware allocated to a base station is often overprovisioned to account for the worst-case scenario. However, this makes it typically challenging to also be energy efficient when there is less load. Therefore, we need adaptive solutions that can react to the current demands and adjust the configuration such that the workload can be handled within the expected bounds while optimizing for energy efficiency. It can be argued that implementing adaptive behavior for base stations requires a formal approach based on a MoC.[44]

In this section, we leverage Mocasin to prototype a tool flow for mapping an LTE baseband processing application using SDF. This prototype presents the foundation for the investigation and evaluation of novel hybrid mapping strategies specifically for the telecommunications domain, as published in Khasanov, Robledo, et al. 2021, *Domain-Specific Hybrid Mapping for Energy-Efficient Baseband Processing in Wireless Networks*.

44: Wittig et al. 2020, *Modem Design in the Era of 5G and Beyond: The Need for a Formal Approach*.

7.3.1 Application Model

Physical layer baseband processing in an LTE base station is a computationally demanding task. Especially in the context of Cloud Radio Access Networks (RANs), parallelization of processes and good mapping strategies are central to an efficient execution that meets the real-time deadlines of the protocol.[45] This subsection describes the SDF model that we use to represent this workload.

The LTE model considered in this section is based on the open-source PHY benchmark,[46] which provides an implementation of an LTE physical layer uplink receiver. From this benchmark, we can extract an SDF model. A central aspect of this model is that the specific size and topology of the SDF graphs depend on the workload they are processing.

LTE transmissions are divided into subframes of 1 ms length. In each subframe, the base station may receive data packets from up to 10 devices, which are called user equipment. Each instance of the SDF model processes the

45: Budhdev, Chan, and Mitra 2020a, *Isoran: Isolation and Scaling for 5g Ranvia User-Level Data Plane Virtualization*; Budhdev, Chan, and Mitra 2020b, *Poster: IsoRAN: Isolation and Scaling for 5G RAN via User-Level Data Plane Virtualization*; Heulot, Boutellier, et al. 2013, *Applying the Adaptive Hybrid Flow-Shop Scheduling Method to Schedule a 3GPP LTE Physical Layer Algorithm onto Many-core Digital Signal Brocessors*; Venkataramani et al. 2020, *Time-Predictable Software-Defined Architecture with SDF-based Compiler Flow for 5G Baseband Processing*.

46: Själander et al. 2012, *An LTE Uplink Receiver PHY Benchmark and Subframe-based Power Management*.

data received from one user equipment within one subframe. Depending on the type and size of data packets received, the size and structure of the SDF graph required for processing this data change. For the workloads that we investigate in this section, the graphs have between 78 and 234 actors and between 1096 and 3228 communication channels. In upcoming technologies, like 5G and beyond, the variability in the workloads is expected to increase further. The interested reader may find a more detailed description of the workload in the literature.[47]

We use the Odroid-XU4 as an evaluation platform for this case study. By measuring the execution times of individual actors in the PHY benchmark on the Odroid-XU4, we can enrich the SDF model that we use in Mocasin with realistic performance characteristics. While general-purpose architectures like the Odroid board are not ideal for baseband processing, they can be utilized for small base stations (e.g., Femtocells).[48] For the purposes of prototyping, this setup allows us to use realistic numbers to assess the general trends and compare our simulated results to a real execution. Modeling a more realistic scenario, including specialized hardware and real LTE workloads, is beyond the scope of this case study.

47: Khasanov, Robledo, et al. 2021, *Domain-Specific Hybrid Mapping for Energy-Efficient Baseband Processing in Wireless Networks*; Robledo and Castrillon 2022, *Parameterizable Mobile Workloads for Adaptable Base Station Optimizations*.

48: Budhdev, Chan, and Mitra 2018, *PR³: Power Efficient and Low Latency Baseband Processing for LTE Femtocells*.

7.3.2 Toolflow

Leveraging Mocasin's configurable infrastructure, we can create a toolflow that extrapolates from the single SDF graph to simulate the processing of a continuous stream of incoming data. Figure 7.8 shows the overall architecture of this toolflow. It is implemented as a Mocasin plugin and is available separately.[49]

The plugin provides two additional modules: the *workload generator* and the *LTE simulation manager*. The workload generator continuously reads data from a workload description file. It models the antenna and continuously produces new SDF graphs and traces to handle the incoming data. In this case study, we only consider randomized synthetic workloads, but generally, workloads can also be recorded from traffic observed at real base stations[50] or generated from models of real workloads.[51]

The LTE simulation manager hooks into Mocasin's simulator to resemble a dynamic that maps new SDF applications as the workload generator creates them. Every 1 ms (in simulated time), the LTE simulation manager requests the workload for a new subframe. The simulator passes information about the current system state (e.g., load) to the mapper and receives mappings for new applications arriving at the current subframe. Many of the generated

49: Menard, Robledo, and Khasanov 2023, *Fivegsim: A Simualtor for 5G Baseband Applications Based on Mocasin*.

50: Budhdev, Chan, and Mitra 2020a, *Iso-ran: Isolation and Scaling for 5g Ranvia User-Level Data Plane Virtualization*.

51: Robledo and Castrillon 2022, *Parameterizable Mobile Workloads for Adaptable Base Station Optimizations*.

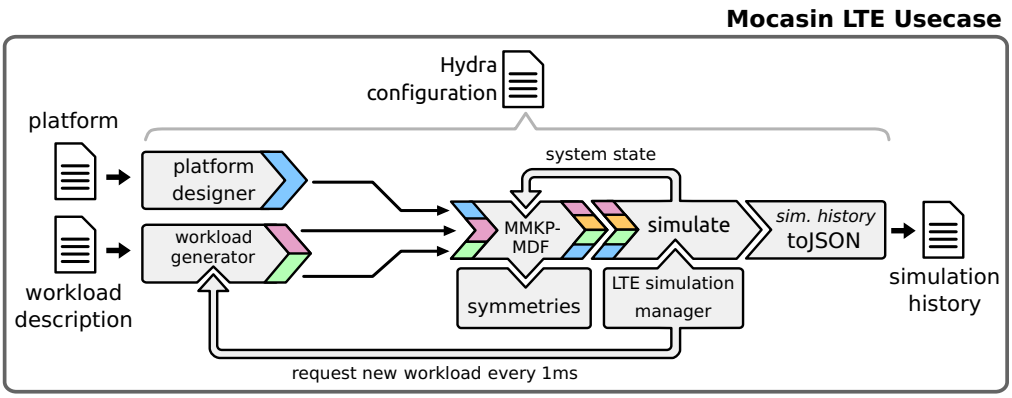


Figure 7.8: A Mocasin toolflow for mapping and simulating the workload of an LTE uplink receiver.

SDF applications will run for more than 1 ms. Thus, newly created SDF applications have to share resources with the ones still executing from previous subframes. Consequently, a good mapping strategy should consider the current load of the system and the mappings of applications already running.

In Figure 7.8, the MKKP-MDP algorithm[52] is used for mapping applications to the Odroid architecture. This mapping algorithm implements a hybrid mapping strategy that considers the current system state. It is based on TETRiS, which leverages the symmetry representation module to reduce the size of the mapping space.[53] Since all mappers in Mocasin implement the same interface, we can leverage any of the existing mappers to generate mappings on the fly and prototype new strategies tailored specifically for the LTE use case. The Fivegsim plugin elevates Mocasin's simulator from a tool for estimating the performance of static mappings to a tool for researching dynamic runtime strategies for mapping an LTE workload.

52: Khasanov and Castrillon 2020, *Energy-efficient Runtime Resource Management for Adaptable Multi-application Mapping*.

53: Goens, Khasanov, Hähnel, et al. 2017, *TETRiS: a Multi-Application Run-Time System for Predictable Execution of Static Mappings*; Goens, Siccha, and Castrillon 2017, *Symmetry in Software Synthesis*.

To validate the system model, we compared the high-level simulation in Mocasin with the real execution of the PHY benchmark on the Odroid-XU4 for the same workload. Figure 7.9 shows the results of this comparison. The red line indicates ideal behavior, where the measured and simulated times coincide, whereas the blue dotted line shows the result of a linear regression. The diagram shows that the simulated execution time is slightly inaccurate. The error, however, is systematic. The simulation is off by approximately a factor of 1.4. Such a systematic error is less problematic, as the main motivation for creating the toolflow is to analyze and compare mapping/scheduling approaches, not to simulate accurate timings. Thus, a better strategy for assessing the quality of the results is to evaluate the fidelity of the simulation. In terms of fidelity, a linear regression yields a p-value $< 10^{-15}$ and the data also features a high Spearmann's correlation of $\rho = 0.978$. This indicates that we can reliably compare the effects of various mapping strategies in Mocasin, since a lower estimated simulation time also indicates a better performance on the real platform.

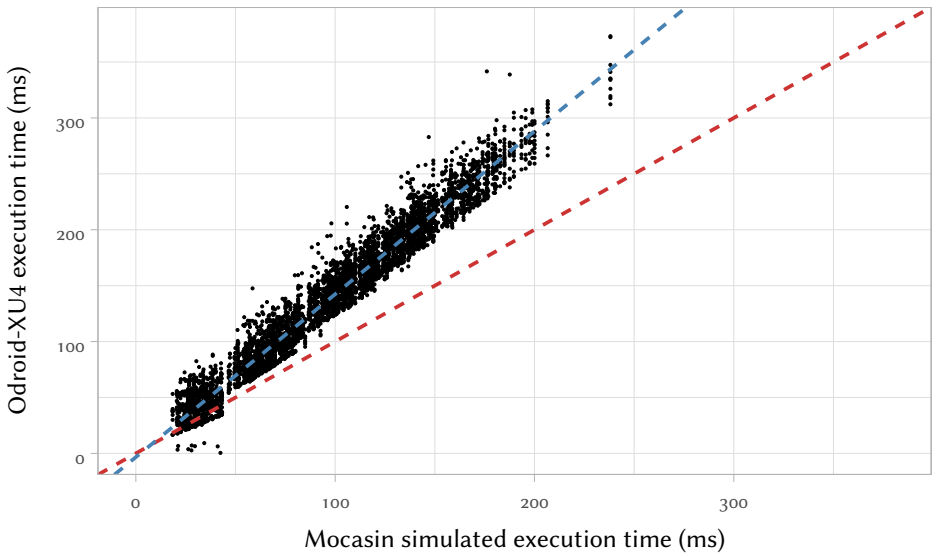


Figure 7.9: Validation of the LTE simulation in Mocasin. Every point represents the simulated execution time and its corresponding measured execution time for a random workload.

7.3.3 Evaluating Mapping Strategies

Using the newly created toolflow, we can investigate how best to cope with the dynamic workload of LTE baseband applications. For this, we perform an experiment that compares the quality of mappings produced by different algorithms for a range of workloads. Most of the mappers implemented in Mocasin assume a single application with a static structure (which possibly consists of multiple tasks, processes, or actors) and are intended to be used once during deployment. The newly created toolflow, however, enables using the same mapping strategies to mimic a runtime strategy that handles multiple applications. Clearly, these algorithms are not designed to be used at run-time. However, this method enables establishing a baseline for further research on runtime heuristics.[54]

54: Khasanov, Robledo, et al. 2021, *Domain-Specific Hybrid Mapping for Energy-Efficient Baseband Processing in Wireless Networks*.

To evaluate the existing mapping strategies, we generate random Poisson-distributed LTE workloads and compare the performance of the benchmark using different mapping algorithms. Baseband processing in LTE is a firm real-time application—after the real-time deadlines have passed, the results are useless. We model this by terminating a running application once the deadline of 2.5 ms has passed and recording the deadline miss. Figure 7.10 shows the miss rates recorded in our simulated prototype for a range of mapping algorithms and for two scenarios with comparatively lower and higher workloads.

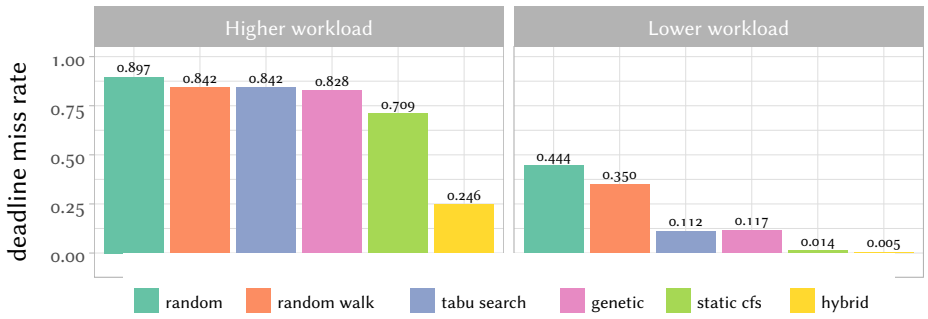


Figure 7.10: Comparison of various mapping algorithms applied to the randomized workloads of the LTE benchmark.

The plot shows that both the random and random walk strategies perform badly. For the lower workload, the meta-heuristics genetic and tabu search perform significantly better. However, out of the static mapping approaches, the static CFS mapper performs best. This is because we parameterized the meta-heuristics for quick execution, limiting the number of generations, and because these meta-heuristics struggle generally with a large number of actors as the mapping space grows exponentially. Since the dependencies between nodes in the SDF graph are regularly structured and the number of actors is relatively large, a load-balancing strategy as realized by the static CFS mapper seems to work best. This is different from applications with more coarse-grained actors and complex interdependencies, where meta-heuristics significantly outperform static CFS. However, the results also suggest that the hybrid mapping approach (concretely, the MKKP-MDP algorithm),[55] which considers the current system load, produces better mappings than all static approaches.

55: Khasanov and Castrillon 2020, *Energy-efficient Runtime Resource Management for Adaptable Multi-application Mapping*.

Overall, this case study illustrates that new toolflows can be prototyped and evaluated quickly in Mocasin. Based on the exploration performed on the LTE use case, we can conclude that hybrid strategies work better for the LTE use case, even compared to the computationally costly static mapping strategies.

The prototype created in Mocasin provides a baseline for researching more elaborate mapping strategies for LTE workloads.[56]

56: Khasanov, Robledo, et al. 2021, *Domain-Specific Hybrid Mapping for Energy-Efficient Baseband Processing in Wireless Networks*.

7.4 Integrating Mocasin with Lingua Franca

While Mocasin was initially developed to support research in the domain of DSE for dataflow applications, it can in principle also be used for researching DSE for other MoCs. Most DSE tools hard-code assumptions about the MoC and the toolflow itself. Mocasin, however, embraces a modular and flexible architecture that allows for prototyping new tools. The previous section, for instance, showed that we can use Mocasin to simulate hybrid mapping strategies, although this was not part of Mocasin's original design.

In principle, Mocasin also allows for incorporating the reactor model. Such an integration would provide the ideal foundation for researching DSE toolflows for reactors and Lingua Franca. In this section, we discuss how Mocasin could be used for constructing DSE flows based on the reactor model. This discussion, however, only includes proposals and potential solutions. While we have performed some early experimentation, fully integrating reactors and LF with Mocasin remains for future work.

7.4.1 Static Subsets

One option for modeling reactor programs in Mocasin is to convert them to SDF or KPN programs. However, as Section 2.4.4 argues, dataflow models and process networks are not suitable for modeling timed or reactive behavior like it is possible with reactors. Therefore, we cannot derive equivalent SDF or KPN models for general reactor programs. There are, however, subsets of reactors and LF that can be represented as SDF programs.

For any program that only uses timers and ports, we know at compile time precisely which events will be present. In fact, we can derive a static schedule for executing such applications. This subset of reactors does not model reactive behavior, and we can also represent the behavior in SDF. Thus, we can trivially apply Mocasin to map such programs to heterogeneous hardware.

However, restricting programs to using timers and ports and forbidding the use of actions is a severe limitation. In principle, some more general programs using actions can also be scheduled statically if we constrain the delay imposed when scheduling the action. Shaokai Lin at UC Berkely is currently researching the limits of static schedulability for LF programs. However, programs using physical actions without any further constraints on when new events may arrive or logical actions without constraining the delays that the reaction code may choose cannot be scheduled statically and cannot be trivially represented in SDF or KPN.

7.4.2 Replaying Traces for equivalent KPNs

While a general reactor program cannot be trivially represented as a KPN program, we can instead only focus on one particular execution. Given a trace that represents the execution of a reactor program, including all the events and the reactions executed in response to these events, we can generate a KPN graph and trace that resembles a similar behavior.

The main challenge when implementing reactive behavior in KPN is that the receiver of a message needs to know in advance how many tokens it may expect on a channel. If the process reads more tokens than the sender produces, it may block indefinitely. Thus, to avoid deadlocks, there needs to be some knowledge about expected communication patterns.

Given a trace for the execution of a reactor program, we know precisely the communication pattern of this single execution, and we can create a representation for this single behavior in Mocasin's KPN model. For this, we create a KPN graph, where the processes denote reactions and the edges denote dependencies between reactions. Then we can synthesize a trace for this KPN application that replays the execution of the reactor program.

While early experimentation conducted by Anton Landgraf confirms that this approach works in principle, the fundamentally different semantics of reactors and KPN substantially limit the accuracy that can be achieved with this approach.

7.4.3 Implementing the Reactor MoC in Mocasin

Most likely, the most promising approach for modeling reactor programs in Mocasin is to extend Mocasin's simulation module with an implementation of the reactor MoC. This is challenging, however. As discussed in Section 7.2.4, Mocasin's simulator uses a process model as an interface between the simulator core and the semantics of the MoC that can be implemented on top of it. This works well for MoCs that can be described as multiple interacting processes, like KPN. Reactor programs, however, are implemented as single processes that may potentially utilize multiple worker threads. To model reactor execution accurately in Mocasin, we need to extend the simulator and introduce concepts for simulating multi-threading within a process.

The process model currently implemented in Mocasin, however, can more accurately represent federated programs or programs that use enclaves. Enclaves and federates can be naturally represented as multiple interacting processes. Assuming that we can simulate the execution of one reactor program in a single process, we can extend this model by implementing a coordination protocol that resembles the federated or enclave coordination schemes. This model would be sufficient for leveraging Mocasin's existing mapping algorithms for allocating resources to the enclaves or federates. Simulating the coordination protocol in Mocasin would not only be useful for performing DSE, but it would also provide a platform for prototyping and analyzing new coordination mechanisms.

7.4.4 Automatic Partitioning

Given that we can simulate the execution of reactor programs consisting of multiple enclaves in Mocasin, we could also consider using Mocasin for automatically partitioning LF programs. This, however, adds another dimension to the mapping problem. Mocasin's mappers are concerned with assigning hardware resources to processes. Partitioning a program, however, requires mapping atomic elements (i.e., reactors) of the input model to processes. This will require a different set of mapping heuristics. Most likely, the most effective algorithms would jointly consider both partitioning and mapping. Addressing the problem of automatically partitioning and mapping reactor programs is a novel problem, and solving it potentially opens entirely new directions for research. An integration of the reactor model in Mocasin would facilitate this research and provide a platform for experimentation.

7.5 Conclusion

This chapter introduced the general concepts behind DSE and introduced the Mocasin framework, which is a core contribution of this thesis. In contrast to existing DSE tools, Mocasin itself is designed as an exploration platform for research in the domain of DSE, in particular for mapping applications to heterogeneous target architectures. The presented case study, based on an LTE baseband application, demonstrates Mocasin's ability to prototype new toolflows for researching particular DSE problems. The modular architecture of Mocasin, in principle, also allows for integrating reactors and Lingua Franca. Such an integration would create a research platform for developing reactor-specific mapping strategies as well as heuristics for automatic partitioning. While Section 7.4 proposed several approaches for integrating reactors with Mocasin, implementing a full integration and conducting research on reactor-specific DSE strategies remain for future work.

8 Related Work

This chapter provides a brief overview of other publications and tools that are related to the work presented in this thesis. Most of the referenced works are discussed in more depth in other chapters of this thesis. Therefore, this chapter presents a summary and references to more detailed discussions in other parts of this thesis.

8.1 Models of Computation

The reactor model, initially defined by Lohstroh, Romeo, et al., presents the foundation for this thesis.[1] The reactor model is closely related to a range of existing MoCs and borrows concepts from them. For instance, reactors resemble Hewitt actors[2] in that they encapsulate state and communicate with other reactors via messages. In contrast to Hewitt actors, however, the execution semantics of reactors is deterministic. The communication of reactors based on ports and connections is similar to communication in SDF[3] and KPN.[4] The synchronous semantics of communication in reactors, however, resembles the signals known from synchronous languages.[5] Finally, the discrete event semantics at the core of the reactor model establishes a logical timeline and enables reactive behavior.

1: Lohstroh, Romeo, et al. 2019, *Reactors: A Deterministic Model for Composable Reactive Systems*; Lohstroh 2020, *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*.

2: Hewitt, Bishop, and Steiger 1973, *A Universal Modular ACTOR Formalism for Artificial Intelligence*.

3: E. A. Lee and Messerschmitt 1987, *Synchronous Data Flow*.

4: Kahn 1974, *The Semantics of a Simple Language for Parallel Programming*.

5: Benveniste, Caspi, et al. 2003, *The Synchronous Languages 12 Years Later*.

The reactor model combines the strengths of the aforementioned models while avoiding some of their pitfalls. Chapter 2 provides an in-depth survey of the related MoCs as well as their limitations regarding CPS design. It argues that none of the existing MoCs are deterministic, reactive, timed, *and* scalable. This thesis demonstrates that this semantic gap can be closed by the reactor model and supporting tools.

Section 6.5 argues that the semantics of some MoCs can be replicated using reactors. In particular, the nondeterministic behavior of Hewitt actors can be replicated using physical actions, and the behavior of LET[6] can be replicated using logical delays on connections.

6: Kirsch and Sokolova 2012, *The Logical Execution Time Paradigm*.

8.2 Languages and Frameworks

There is a wide range of programming languages and frameworks that are related to this thesis and, in particular, to Lingua Franca. Perhaps the closest in spirit is the Ptolomy II project.[7] Ptolemy II has a strong focus on modeling CPS and significantly influenced the design and development of Lingua Franca. Ptolemy II provides implementations for many of the MoCs discussed in this thesis and it allows for combining different MoCs hierarchically. However, the fundamental actor semantics that Ptolemy II assumes as a blueprint for each of the MoCs[8] does not well resemble the semantics of reactors. Moreover, Ptolemy II focuses on modeling and simulating CPS, while Lingua Franca provides a language for implementing software that can be deployed to real systems.

7: Ptolemaeus 2014, *System Design, Modeling, and Simulation using Ptolemy II*.

8: Tripakis et al. 2012, *A Modular Formal Semantics for Ptolemy*.

Ptolomy II uses a graphical approach for composing models, similar to commercial tools like LabVIEW[9] and Simulink.[10] While Lingua Franca also provides a graphical representation of reactor programs, it follows a different philosophy and prioritizes a textual representation. The textual LF

9: Bitter, Mohiuddin, and Nawrocki 2017, *LabVIEW: Advanced Programming Techniques*.

10: Dabney and Harman 2004, *Mastering Simulink*.

program represents the ground truth. The diagrams are generated from this textual description and provide another view that is useful for studying and navigating a program, but cannot be used for modifying the program.

ROS[11] and AUTOSAR AP[12] are closely related to LF in their scope. Both aim to provide a foundation for developing complex CPS applications. However, as discussed in Section 2.9, their underlying communication paradigms, namely publish/subscribe[13] and SoA,[14] are semantically equivalent to Hewitt actors and expose the same nondeterministic behavior. NASA develops another framework for embedded systems design called F′ (F prime)[15] that is structurally similar to Lingua Franca. While F′, to some extent, can also make guarantees about the order in which events are observed by components, these guarantees are less strong than in Lingua Franca.

There are many languages and frameworks that implement the Hewitt actor model and that are used for systems design (cf. Section 2.3). However, due to the exposed nondeterminism, they are not well suited for CPS design, where it is crucial to understand and test the precise behavior. Reactive programming frameworks like ReactiveX[16] and Raectors.IO[17] are also closely related to LF, but they share the problems of Hewitt actors and require the programmer to explicitly introduce synchronization.

In the domain of more classical real-time systems, Ada[18], Real-Time Euclid[19] and Real-time Java[20] are noteworthy programming languages. However, these languages focus on time predictability, and, as discussed in Section 2.2.4, the underlying real-time task model is nondeterministic. Timed C[21] is more closely related to Lingua Franca, as it also introduces a logical timeline and enforces a deterministic execution. However, Timed C is limited to the C programming language and provides a task-based programming model that does not expose any hierarchical components or an explicit notion of dependencies. Giotto is another language that supports the definition of deterministic programs as a composition of hard real-time tasks based on the LET paradigm.[22] However, Giotto is also limited to a task model, and unlike LF, it does not support reactions to sporadic events.

Hardware description languages and discrete event simulation frameworks are closely related to LF (cf. Section 2.6.1) for two main reasons. Firstly, they commonly facilitate a notion of components that bears a resemblance of reactors in LF, and secondly, they build on top of a discrete events semantics. Particularly noteworthy is Bluespec,[23] as it uses guarded atomic actions for defining hardware, an approach very similar to the reactions of LF. However, such languages and frameworks are designed for simulating and synthesizing hardware, not for executing software. Furthermore, many of the available tools expose nondeterminism, as there is no well-defined order for simultaneous events.

The family of synchronous programming languages (cf. Section 2.7) is also closely related to LF. In particular, SCADE[24] is commercially successful and has been applied to various safety-critical industrial use cases. However, the synchronous-reactive model exposes concurrency at a fine-granular level and is difficult to scale up to large systems. In this context, the sparse synchronous model, as implemented in Scoria,[25] is a promising extension of the synchronous reactive paradigm. It allows for describing more coarse-grained concurrent processes that are forked off and joined at well-defined logical times. Such concurrent processes impose a logical delay similar to LET tasks. Thus, Scoria, like LF, can describe both fully synchronous behavior and LET-like processes that impose logical delays. Recently, the concept of *logical synchrony* was proposed as a framework for coordinating distributed execution based on the synchronous-reactive paradigm. It introduces a similar notion of "logical latency."[26]

11: Quigley et al. 2009, *ROS: an Open-source Robot Operating System*; Koubaa 2016, *Robot Operating System (ROS)—The Complete Reference (Volume 1)*.

12: AUTOSAR 2023, *AUTomotiv Open System ARchitecture*.

13: Eugster et al. 2003, *The Many Faces of Publish/subscribe*.

14: Perrey and Lycett 2003, *Service-oriented Architecture*; Papazoglou and Heuvel 2007, *Service Oriented Architectures: Approaches, Technologies and Research Issues*; K. B. Laskey and K. Laskey 2009, *Service Oriented Architecture*.

15: Bocchino et al. 2018, *F Prime: an Open-Source Framework for Small-Scale Flight Software Systems*.

16: Meijer 2010, *Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues*.

17: Prokopec 2018, *Pluggable Scheduling for the Reactor Programming Model*.

18: Burns and Andy Wellings 2007, *Concurrent and real-time programming in Ada*.

19: Kligerman and Stoyenko 1986, *Real-Time Euclid: a Language for Reliable Real-Time Systems*.

20: Andrew Wellings 2004, *Concurrent and Real-time Programming in Java*.

21: Natarajan and Broman 2018, *Timed C: An Extension to the C Programming Language for Real-Time Systems*.

22: Henzinger, Horowitz, and Kirsch 2003, *Giotto: A Time-Triggered Language for Embedded Programming*.

23: Arvind et al. 2004, *High-level Synthesis: an Essential Ingredient for Designing Complex ASICs*.

24: Abdulla et al. 2006, *Designing Safe, Reliable Systems Using Scade*; Gérard Berry 2007, *SCADE: Synchronous Design and Validation of Embedded Control Software*; Colaço, Pagano, and Pouzet 2017, *SCADE 6: A formal language for embedded critical software development (invited paper)*.

25: Krook et al. 2022, *Creating a Language for Writing Real-Time Applications for the Internet of Things*.

26: Lall et al. 2023, *Logical Synchrony and the Bittide Mechanism*.

Some streaming languages and frameworks build on similar ideas, and utilize logical clocks to coordinate distributed execution deterministically. Timely dataflow, for instance, is a framework that allows for expressing general streaming applications in Rust. It coordinates the execution locally or on distributed machines using vector clocks and special timestamps that encode information about the causal relationship of events.[27] The Hydroflow Rust framework builds on similar ideas.[28] However, the notion of time in these frameworks is purely logical and is not exposed to the programmers. Thus, it cannot be used for reasoning about events on a physical timeline. Moreover, while Hydroflow and Timely dataflow ensure determinism by preserving causality, they cannot easily reason about the correct order of input events when their causal relationship is defined outside the system.

27: Murray et al. 2013, *Naiad: A Timely Dataflow System*.

28: Samuel 2021, *Hydroflow: A Model and Runtime for Distributed Systems Programming*.

Ohua[29] and ConDRust[30] follow a different approach and automatically derive parallel dataflow implementations from sequential algorithms. The compiler analyzes the data and control-flow dependencies between the statements of the sequential program and automatically derives a graph that captures these dependencies (similar to the APG in LF). This graph is then compiled into a parallel dataflow application that preserves the semantics of the sequential program. This approach, however, only provides indirect control over the precise composition of the program and does not provide a notion of time as part of its semantics.

29: Ertel 2019, *Towards Implicit Parallel Programming for Systems*.

30: Suchert et al. 2023, *ConDRust: Scalable Deterministic Concurrency from Verifiable Rust Programs*.

8.3 Scalable Connection Patterns and Performance Optimization

The concept of banks and multiports introduced in Section 5.1 resembles similar concepts in the Ptolemy II project. Ptolemy II also supports multiports and provides some higher-order components that allow for creating multiple parallel copies of the same actor. However, Ptolemy II does not provide a textual syntax for these concepts. More recently, CAL introduced port arrays, which are similar to multiports in LF.[31]

31: Callanan and Gruian 2023, *Scalable Actor Networks with CAL*.

The general implementation of the C++ runtime and the optimizations discussed in Section 5.2 are unique to Lingua Franca. While conceptually related, scheduling reactor programs is fundamentally different from scheduling Hewitt actors and related MoCs. Since Hewitt actors are nondeterministic and the workload cannot be predicted, the runtime scheduler needs to make ad hoc decisions to distribute the workload. The predominant solution is work stealing,[32] which is also the default scheduling mechanism of Akka and CAF. The main advantage of work stealing is that it avoids a centralized scheduler and minimizes the synchronization points between workers. As long as they have sufficient work, workers can operate independently. The work stealing approach, however, does not work well for a reactor runtime, as deciding which reactions are ready to process requires more knowledge about the system's state. While the reactor MoC requires a central scheduler, we can better leverage knowledge about the program and its dependencies to optimize the execution.

32: Blumofe and Leiserson 1999, *Scheduling Multithreaded Computations By Work Stealing*; Yang and He 2018, *Scheduling Parallel Computations By Work Stealing: a Survey*.

Since the reactor model is based on discrete events, the presented scheduling algorithm is closely related to the mechanisms used in discrete event simulators (cf. Section 2.6.1). However, parallelizing the execution in such simulators is commonly hard as the dependencies and the precise interactions of components are not known in advance. The scheduler presented in Section 5.2 instead leverages the properties of the reactor model to understand precisely which reactions can be executed in parallel to deliver an efficient, deterministic execution.

8.4 Design Space Exploration

The Mocasin framework introduced in Section 7.2 is closely related to a wide range of existing DSE frameworks described in the literature (cf. Section 7.1). However, in contrast to the existing tools, Mocasin provides a research platform for prototyping toolflows and experimenting with novel DSE concepts. It is a complementary tool designed to facilitate research on DSE tools in general. To achieve comparability with existing approaches, Mocasin incorporates many mapping strategies known from the literature into its toolbox.

Ptolemy II follows an idea very similar to Mocasin—to provide a rapid prototyping environment independent of a particular use case to facilitate research and development. However, Ptolemy II focuses on researching and experimenting with actor-based MoCs and researching new MoCs. It accurately simulates the application behavior according to the MoC semantics. While Ptolemy II focuses on accurately capturing the semantics and functional properties of applications, Mocasin completely abstracts over the semantics and only considers application properties that are relevant for rapid performance estimation. Mocasin is complementary in the sense that Ptolemy II is a toolbox for creating accurate models of applications, and Mocasin is a toolbox for creating DSE flows for generating efficient implementations of given applications on a wide range of hardware architectures.

Conclusions 9

This chapter presents a summary of the contributions made in this thesis and discusses possible directions for future work.

9.1 Summary

This thesis introduces a complete methodology and comprehensive software stack supporting the design and development of deterministic reactive programs for cyber-physical systems. At the core of this methodology is the novel reactor model. In contrast to related models, the reactor model promises to fulfill the main objectives of this thesis as defined in Chapter 1: *timed semantics*, *reactivity*, *scalability*, and *determinism*.

The presented stack of tools includes the DEAR framework, which constitutes one of the first practical implementations of the reactor model and integrates it with AUTOSAR. The conducted case study based on the AUTOSAR Adaptive Platform Demonstrator exposes the problem of nondeterminism in safety-critical applications. It also highlights the general applicability of reactors to industrial use cases and the ability of reactors to achieve *determinism* even in distributed applications.

Building on the reactor model, this thesis contributes to the coordination language Lingua Franca, which enables a more efficient definition of reactor programs by providing an abstract textual syntax. This includes the development of a full-fledged C++ reactor runtime based on the DEAR framework, as well as a comprehensive LF code generation backend. The overall discussion in Chapter 4 highlights LF's ability to coordinate execution across a multiplicity of timelines, and the presented code examples demonstrate how we can utilize the *timed semantics* and *reactivity* of reactors to solve relevant problems.

This thesis also demonstrates the *scalability* of LF applications in a comprehensive evaluation (cf. Chapter 5). The comparison with the popular actor frameworks Akka and CAF reveals that LF can significantly outperform state-of-the-art actor frameworks, in particular if the workloads require some form of synchronization. However, Chapter 6 also shows that there are some limitations to this scalability that result from the constraints currently imposed by the reactor scheduler. This thesis proposes scheduling enclaves as a mechanism for partitioning LF programs to better decouple different parts of the program and enable their parallel execution.

Another aspect of *scalability* is the ability of a toolflow to manage hardware complexity. Chapter 7 argues the case for leveraging DSE techniques to automate the process of deploying an application to a concrete realization on complex hardware. It also introduces the Mocasin tool as a research platform in the DSE domain. While an integration of Mocasin with LF remains for future work, the scope and modular structure of Mocasin present an ideal foundation for experimenting with state-of-the-art DES techniques in LF.

In conclusion, this thesis introduces a reactor-based workflow that indeed closes the semantic gap identified in Section 2.10. The coordination language Lingua Franca makes reactor-based programming more accessible

and provides comprehensive tooling that supports the development of complex applications. The C++ runtime provides a particularly well-performing implementation of the reactor model that can compete with and even exceed the performance of popular actor frameworks.

9.2 Future Work

While this thesis makes a significant contribution to the methodologies and tooling available for CPS design, the discussion also touches on various limitations of the proposed approach, which opens interesting opportunities for future work.

9.2.1 Coordination of Enclaves and Federates

The mechanisms for coordinating federates and enclaves that are currently implemented in LF all have significant limitations. The centralized coordination scheme for federated execution requires a central coordinator, which presents a single point of failure and a performance bottleneck. Decentralized coordination scales better but instead requires assumptions about clock synchronization error, network latency, and WCET. In particular, good WCET optimizations are challenging to obtain since the complexity of embedded hardware increases drastically, which often decreases time predictability. WCET estimations often need to be overly conservative, which leads to long waiting times.

The mechanism for coordinating enclaves introduced in Section 6.3 aims to provide an alternative coordination scheme that allows neighboring reactors to directly coordinate between themselves without an additional coordinator. However, while this approach works well for simple programs, this coordination scheme can become very inefficient if the reactor graph contains cycles. Moreover, cycles between reactors without any delays are not supported at all.

Improving the existing coordination mechanisms requires more research. Likely, different coordination mechanisms could be combined to achieve the best results. While acyclic structures could coordinate execution directly without resorting to a coordinator, reactors in a cyclic structure could use a local coordinator that governs only a subset of the overall program that is part of the cycle. Furthermore, related work could provide inspiration on how to improve the existing scheduling mechanisms. In particular, the methodologies of Timely dataflow[1] and bittide[2] are relevant in the context of LF. Both achieve deterministic coordination without a central coordinator, and both use logical timestamps to reason about the order of events.

1: Murray et al. 2013, *Naiad: A Timely Dataflow System*.

2: Lall et al. 2023, *Logical Synchrony and the Bittide Mechanism*.

Further research on coordinating reactor programs should also extend the benchmarks presented in this thesis to LF programs using federates or enclaves and evaluate the performance of implemented approaches.

9.2.2 Design Space Exploration in LF

There are two major avenues of DSE in Lingua Franca. First, DSE techniques could be utilized for mapping reactors and reactions to particular hardware resources. And, second, DSE techniques could be utilized to automatically partition LF programs and introduce enclaves where needed to achieve certain requirements. This could be ensuring that parts of the application

can effectively utilize parallel resources to achieve a certain throughput or that dependencies are broken up such that deadlines are met. Currently, it is unclear if this process can be completely automated. However, even if deriving good heuristics proves challenging, DSE techniques can provide insights and guidance for users while developing the program.

The Mocasin DSE tool presents one of the core contributions of this thesis. Mocasin was designed specifically to provide a modular research platform for prototyping DSE flows. While Mocasin focuses on modeling dataflow applications, the modular architecture, in principle, also allows for an integration of the reactor model. Section 7.4 discusses a few possible paths for integrating LF with Mocasin. This integration, however, as well as the development of concrete DSE flows and heuristics, remain for future work.

9.2.3 Mutations

The reactor model defines mutations, which allow a reactor to modify itself during execution. Mutations could, for instance, instantiate new reactors or establish new connections. They could also disconnect or deconstruct reactors. Mutations are the reactor equivalent of the dynamic instantiation of new actors in the Hewitt actor model. Mutations modify the structure of the program and, consequently, also the dependencies between reactions recorded in the APG.

At the moment of this writing, Lingua Franca does not fully support mutations. For this reason, the performance evaluation in Section 5.3 omits Savina benchmarks that require dynamic actor creation. There are multiple challenges to implementing mutations in LF that need to be resolved in future work. Besides providing a syntax for expressing mutations, their scope and capabilities need to be defined precisely. Mutations can be a powerful tool for building adaptable software, but when used excessively, they make programs difficult to understand and analyze. There is a trade-off between the expressiveness of mutations and the overall predictability and analyzability of programs, and an LF implementation needs to provide a well-balanced solution.

On the side of the runtime, mutations require support for changing the APG during execution. Currently, the reactor runtimes assume that the APG only has to be calculated once. While, in principle, the entire APG can be recomputed in between two tags, this process would likely introduce significant overhead for sufficiently large graphs. Therefore, mutations should be limited in scope so that only subsets of the APG have to be reconsidered.

9.2.4 Hardware Synthesis

As mentioned in Section 8.2, LF bears some resemblance to HDLs. Similar to real hardware circuits, reactions respond to incoming signals and may change outgoing signals. Possibly, LF could also be applied to designing hardware. Reactions themselves could be written in an HDL. Alternatively, high-level synthesis[3] could be leveraged to convert sequential reaction bodies into hardware that performs the same operation as sequential code. However, while there are some parallels to languages used in hardware design, the semantics of the reactor model also deviates significantly from common hardware models. For instance, the scheduling of future events and ordering constraints between reactions (i.e., the APG) could be difficult to realize efficiently in hardware. Exploring the applicability of LF for hardware synthesis remains for future work.

3: McFarland, Parker, and Camposano 1990, *The High-Level Synthesis of Digital Systems*; Coussy and Morawiec 2008, *High-Level Synthesis: From Algorithm to Digital Circuit*.

9.2.5 Integrating LF with Existing CPS Frameworks

While this thesis shows that Lingua Franca presents a promising alternative for modeling CPS software, in particular due to the underlying deterministic semantics of reactors, LF has to compete with a wide range of existing frameworks. In particular, ROS and AUTOSAR AP are popular and widely adopted in industry. The major industry players maintain large legacy code bases, and often their business models rely on the exchangeability and compatibility promised by using a common standard. LF cannot easily replace these existing frameworks and standards. Instead, a more promising approach would be to integrate reactors and LF into the stack of tools that are already deployed in industry. Future work could, for instance, consider synthesizing AUTOSAR-compatible software components from LF programs. F′ could also be a promising target for such an integration, as the mechanisms for structuring programs in F′ are very similar to those in LF.

9.2.6 Reactor Libraries and Higher-level Reactors

Currently, LF developers need to construct their entire application from scratch. Of course, they can import third-party libraries in the target languages, but there is no standard library in LF that includes commonly used reactors. Creating such a library would enable the construction of more complex programs from basic components.

There is also an opportunity to introduce higher-level reactors in LF. Such reactors could use other reactors as arguments to implement certain patterns. For instance, the `LETTask` and `HewittActor` reactors discussed in Section 6.5 could be implemented as higher-level reactors and included in a common library. In addition, certain connection patterns, like the ones discussed in Section 5.1.2, could be provided as higher-level reactors.

List of Figures

List of Tables

List of Listings

Acronyms

AP Adaptive Platform 34
APD Adaptive Platform Demonstrator 55
APG acyclic precedence graph 44, *see also* dependency graph
API application programming interface 28
ARA Runtime Environment for Adaptive Applications 34
AST abstract syntax tree 85
AUTOSAR AUTomotive Open System ARchitecture vii, 7, 8, 33, 34

BET bounded execution time 32

CAF C++ Actor Framework 15
CAL CAL actor language 22
CFS Completely Fair Scheduler 145
CMOS complementary metal-oxide-semiconductor 27
CP Classic Platform 33, 34
CPN C for process networks 22
CPS cyber-physical system vii, 1
CSDF cyclo-static dataflow 22

DAG directed acyclic graph 13
DDF Dennis dataflow or dynamic dataflow 21
DDS Data Distribution Service 18
DEAR Discrete Events for AUTOSAR vii, 7
DES discrete event simulation 28
DMA direct memory access 140
DPN dataflow process network 21
DRAM dynamic random-access memory 140–142
DSE design space exploration vii, 8
DSL domain-specific language 22
DSP digital signal processor 31
DTM deterministic multi-threading 11

EBA emergency brake assistant 23
ECU electronic control unit 3
EMF Eclipse Modeling Framework 85

FFT fast Fourier transformation 20
FIFO first-in, first-out 20
FPGA field-programmable gate array 135

GPU graphics processing unit 135
GUI graphical user interface 138

HDL hardware description language 28
HLA High Level Architecture 81
HPC high-performance computing 6
HSDF homogeneous SDF 21

IDE integrated development environment 85

JVM Java Virtual Machine 105, 110

KPN Kahn process network 19

LET logical execution time 32
LF Lingua Franca vii, viii, xiv, 7
LOC lines of code 5
LSP Language Server Protocol 85
LTE long-term evolution 135, 146

MAPS MPSoC Application Programming Studio 137
MoC model of computation vii, 2
MPSoC multiprocessor system on a chip 28
MQTT Message Queuing Telemetry Transport 18

NoC network-on-chip 140

OOP object-oriented programming 63
OS operating system 34

πSDF parameterized and interfaced SDF 22
POSIX Portable Operating System Interface 34
PTIDES Programming Temporally Integrated Distributed Embedded Systems 52
PTP Precision Time Protocol 84

RAN Radio Access Network 146
ROS Robot Operating System vii, 7, 34
RPC remote procedure call 18
RTI run-time infrastructure 81
RTL register transfer level 28

SADF scenario-aware dataflow 22
SC synchronous constructive 31
SDF synchronous dataflow 21
SDF^3 SDF for Free 141
SoA service-oriented architecture 18
SOME/IP Scalable Service-Oriented Middleware over IP 35
SSM sparse synchronous model 32
SWC software component 35

TDL Time Definition Language 33
TGFF Task Graphs for Free 141
TSN Time-Sensitive Networking 84
TTA time-triggered architecture 33

UML Unified Modeling Language 89

WCET worst-case execution time 11

Symbols

Notation	Description	Page List
c	maximum clock synchronization error	52
d	delay	43
D	deadline	70
$\mathscr{D}$	function that maps a tag and a delay to a tag that is offset by the given delay	43
$\mathfrak{D}_D$	function that maps a physical reading of time to the latest tag that may be executed at this time without violating the deadline D.	70
e	event	25
g	tag	26
g_0	startup tag	43
$\mathscr{G}$	function that maps a timestamp to a tag with microstep o	68
$\mathbb{G}$	set of all tags	26
l	worst-case network latency	52
L	logical execution time	32
m	microstep of a tag	27
M	The largest possible microstep	122
$\mathbb{N}$	set of natural numbers	25
o	offset of a timer	43
p	period of a timer	43
$\mathscr{P}$	function that maps a reading of physical time and a delay to a tag	43
$\mathbb{R}$	set of real numbers	25
s	safe-to-process offset	82
$\mathscr{S}$	function that maps a tag g_d and a delay d to the largest possible tag g with $\mathscr{D}(g, d) = g_d$	122
t	timestamp	25
T	imperfect reading of physical time	25
T_0	imperfect reading of physical time at program start	43
τ	current instance of physical time	25
$\mathscr{T}$	function that maps a tag to its timestamp	26
$\mathbb{T}$	set of all time values	25
w	worst-case execution time	52

Index

Bibliography

Abdulla, Parosh Aziz, Johann Deneux, Gunnar Stålmarck, Herman Ågren, and Ove Åkerlund (2006). *Designing Safe, Reliable Systems Using Scade*. In: *Leveraging Applications of Formal Methods*. Ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 115–129. ISBN: 978-3-540-48929-0 (cited on pages 31, 154).

Abella, Jaume, Carles Hernandez, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega (2015). *WCET Analysis Methods: Pitfalls and Challenges on their Trustworthiness*. In: *10th IEEE International Symposium on Industrial Embedded Systems (SIES)* (cited on page 57).

Advanced Micro Devices, Inc. (2023). *Zynq UltraScale+ MPSoC*. URL: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html (visited on July 13, 2023) (cited on page 135).

Agha, Gul, Ian Mason, Scott Smith, and Carolyn Talcott (1997). *A Foundation for Actor Computation.* In: *Journal of Functional Programming* 7 (cited on page 14).

Alian, Mohammad, Umur Darbaz, Gabor Dozsa, Stephan Diestelhorst, Daehoon Kim, and Nam Sung Kim (2017). *dist-gem5: Distributed simulation of computer clusters*. In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 153–162 (cited on page 29).

Alian, Mohammad, Daehoon Kim, and Nam Sung Kim (2016). *pd-Gem5: Simulation Infrastructure for Parallel/distributed Computer Systems*. In: *IEEE Computer Architecture Letters* 15.1, pp. 41–44 (cited on page 29).

Anand, Madhukar, Arvind Easwaran, Sebastian Fischmeister, and Insup Lee (2008). *Compositional Feasibility Analysis of Conditional Real-Time Task Models*. In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 391–398 (cited on page 12).

Ansys, Inc. (2023). *Ansys SCADE Suite: Model-Based Development Environment for Critical Embedded Software*. URL: https://www.ansys.com/de-de/products/embedded-software/ansys-scade-suite (visited on Sept. 23, 2023) (cited on page 31).

Antinyan, Vard (2020). *Revealing the Complexity of Automotive Software*. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, pp. 1525–1528. ISBN: 9781450370431 (cited on pages 3, 5).

Arditi, Laurent, Amar Bouali, Hedi Boufaied, Gael Clave, Mourad Hadj-Chaib, Laure Leblanc, and Robert de Simone (1999). *Using Esterel and Formal Methods to Increase the Confidence in the Functional Validation of a Commercial DSP*. In: *Proceedings of the ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS)*. Trento, Italy (cited on page 31).

Armstrong, Joe (2007). *A History of Erlang*. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, pp. 6-1-6–26. ISBN: 9781595937667 (cited on page 15).

Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave (2004). *High-level Synthesis: an Essential Ingredient for Designing Complex ASICs*. In: *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* Pp. 775–782 (cited on page 154).

Athas, William C. and Nanette J. Boden (1988). *Cantor: an Actor Programming System for Scientific Computing*. In: *SIGPLAN Not.* 24.4, pp. 66–68. ISSN: 0362-1340 (cited on page 15).

Atlas, Alier and Azer Bestavros (1998). *Statistical Rate Monotonic Scheduling*. In: *Proceedings 19th IEEE Real-Time Systems Symposium*, pp. 123–132 (cited on page 12).

Audsley, Neil, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings (1993). *Applying New Scheduling Theory To Static Priority Pre-Emptive Scheduling*. In: *Software Engineering Journal* 8.5, pp. 284–292 (cited on page 12).

Austad, Henrik and Geir Mathisen (2023). *Bounding the End-to-End Execution Time in Distributed Real-Time Systems: Arguing the Case for Deterministic Networks in Lingua Franca*. In: *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. CPS-IoT Week '23. San Antonio, TX, USA: Association for Computing Machinery, pp. 343–348. ISBN: 9798400700491 (cited on page 84).

AUTOSAR (Mar. 2019). *Guidelines for the Use of the C++14 Language in Critical and Safety-related Systems*. AUTOSAR Standard for Adaptive Platform Document ID 834. Release R19-03 (cited on page 59).

– (Nov. 2022a). *Explanation of Adaptive Platform Design*. AUTOSAR Standard for Adaptive Platform Document ID 706. Release R22-11 (cited on page 34).

AUTOSAR (Nov. 2022b). *Explanation of ara::com API*. AUTOSAR Standard for Classic Platform Document ID 846. Release R22-11 (cited on pages 35, 51).
– (Nov. 2022c). *Methodology for Adaptive Platform*. AUTOSAR Standard for Adaptive Platform Document ID 709. Release R22-11 (cited on page 34).
– (Nov. 2022d). *Methodology for Classic Platform*. AUTOSAR Standard for Classic Platform Document ID 68. Release R22-11 (cited on page 34).
– (Nov. 2022e). *SOME/IP Protocol Specification*. AUTOSAR Standard for Foundation Document ID 696. Release R22-11 (cited on page 35).
– (Nov. 2022f). *SOME/IP Service Discovery Protocol Specification*. AUTOSAR Standard for Foundation Document ID 802. Release R22-11 (cited on page 35).
– (Nov. 2022g). *Specification of Communication Management*. AUTOSAR Standard for Adaptive Platform Document ID 717. Release R22-11 (cited on pages 35, 50).
– (Nov. 2022h). *Specification of Execution Management*. AUTOSAR Standard for Adaptive Platform Document ID 721. Release R22-11 (cited on pages 36, 37).
– (Nov. 2022i). *Specification of Time Synchronization*. AUTOSAR Standard for Classic Platform Document ID 880. Release R22-11 (cited on page 52).
– (Nov. 2022j). *Specification of Timing Extensions*. AUTOSAR Standard for Classic Platform Document ID 411. Release R22-11 (cited on pages 33, 34).
– (2023). *AUTomotiv Open System ARchitecture*. URL: https://autosar.org (visited on Aug. 1, 2023) (cited on pages 34, 154).
Ayguade, Eduard, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang (2009). *The Design of OpenMP Tasks*. In: *IEEE Transactions on Parallel and Distributed Systems* 20.3, pp. 404–418 (cited on page 13).
Bachrach, Jonathan, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović (2012). *Chisel: Constructing Hardware in a Scala Embedded Language*. In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California: Association for Computing Machinery, pp. 1216–1225. ISBN: 9781450311991 (cited on page 28).
Bagherzadeh, Mehdi, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian (2020). *Actor Concurrency Bugs: a Comprehensive Study on Symptoms, Root Causes, Api Usages, and Differences*. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (cited on page 17).
Baker, Henry C. and Carl Hewitt (1977). *The Incremental Garbage Collection of Processes*. In: *SIGPLAN Notices* 12.8, pp. 55–59. ISSN: 0362-1340 (cited on page 18).
Baruah, Sanjoy, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese (2012). *A Generalized Parallel Task Model for Recurrent Real-time Processes*. In: *2012 IEEE 33rd Real-Time Systems Symposium*, pp. 63–72 (cited on page 13).
Baruah, Sanjoy K. (1998). *Feasibility Analysis of Recurring Branching Tasks*. In: *Proceeding. 10th EUROMICRO Workshop on Real-Time Systems*, pp. 138–145 (cited on page 12).
– (2010). *The Non-cyclic Recurring Real-Time Task Model*. In: *2010 31st IEEE Real-Time Systems Symposium*, pp. 173–182 (cited on page 12).
Baruah, Sanjoy K., Deji Chen, Sergey Gorinsky, and Aloysius Mok (July 1999). *Generalized Multiframe Tasks*. In: *Real-Time Systems* 17.1, pp. 5–22. ISSN: 1573-1383 (cited on page 12).
Baruah, Sanjoy K., Aloysius K. Mok, and Louis E. Rosier (1990). *Preemptively Scheduling Hard-real-time Sporadic Tasks on one Processor*. In: *[1990] Proceedings 11th Real-Time Systems Symposium*, pp. 182–190 (cited on page 12).
[SW] Bateni, Soroush, Edward A. Lee, Erling Jellum, Peter Donovan, Marten Lohstroh, Hou Seng Wong, Anirudh Rengarajan, and Chadlia Jerad, *reactor-c*. LIC: BSD-2-Clause. URL: https://github.com/lf-lang/reactor-c (visited on Nov. 10, 2023) (cited on page 87).
Bateni, Soroush, Marten Lohstroh, Hou Seng Wong, Hokeun Kim, Shaokai Lin, Christian Menard, and Edward A. Lee (Sept. 2023). *Risk and Mitigation of Nondeterminism in Distributed Cyber-Physical Systems*. In: *21st ACM-IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE)*, pp. 1–11 (cited on pages x, 37).
Beck, Micah (2019). *On the Hourglass Model*. In: *Communications of the ACM* 62.7, pp. 48–57 (cited on page 2).
Bellassai, Davide, Alessandro Biondi, Alessandro Biasci, and Bruno Morelli (2023). *Supporting Logical Execution Time in Multi-Core Posix Systems*. In: *Journal of Systems Architecture* 144, p. 102987. ISSN: 1383-7621 (cited on page 37).
Benveniste, Albert and Gérard Berry (1991). *The Synchronous Approach To Reactive and Real-Time Systems*. In: *Proceedings of the IEEE* 79.9, pp. 1270–1282 (cited on page 29).

Benveniste, Albert, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert Simone (Feb. 2003). *The Synchronous Languages 12 Years Later*. In: *Proceedings of the IEEE* 91, pp. 64–83 (cited on pages 30, 153).

Berry, Géard and Georges Gonthier (1992). *The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation*. In: *Science of Computer Programming* 19.2, pp. 87–152 (cited on page 78).

Berry, Gérard (May 2000). *The Foundations of Esterel*. In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. The MIT Press. ISBN: 9780262281676 (cited on page 30).

– (2007). *SCADE: Synchronous Design and Validation of Embedded Control Software*. In: *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Ed. by S. Ramesh and Prahladavaradan Sampath. Dordrecht: Springer Netherlands, pp. 19–33. ISBN: 978-1-4020-6254-4 (cited on pages 31, 154).

Berry, Gérard, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, and Robert de Simone (2000). *Esterel: a Formal Method Applied To Avionic Software Development*. In: *Science of Computer Programming* 36.1, pp. 5–25. ISSN: 0167-6423 (cited on page 31).

Berry, Gérard and Ellen Sentovich (2001). *Multiclock Esterel*. In: *Correct Hardware Design and Verification Methods*. Ed. by Tiziana Margaria and Tom Melham. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 110–125. ISBN: 978-3-540-44798-6 (cited on page 30).

Berry, Gérard and Manuel Serrano (2020). *HipHop.js: (A)Synchronous Reactive Web Programming*. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, pp. 533–545. ISBN: 9781450376136 (cited on page 31).

Bilsen, Greet, Marc Engels, Rudy Lauwereins, and Jean A. Peperstraete (1996). *Cyclo-Static Dataflow*. In: *IEEE Transactions on Signal Processing* 44, pp. 397–408 (cited on page 22).

Binkert, Nathan, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood (2011). *The gem5 Simulator*. In: *ACM SIGARCH Computer Architecture News* 39.2. ISSN: 0163-5964 (cited on page 28).

Biondi, Alessandro, Paolo Pazzaglia, Alessio Balsini, and Marco Di Natale (2017). *Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores*. In: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)* (cited on page 33).

Birrell, Andrew D. and Bruce Jay Nelson (1984). *Implementing Remote Procedure Calls*. In: *ACM Transactions on Computer Systems* 2.1, pp. 39–59. ISSN: 0734-2071 (cited on page 18).

Bitter, Rick, Taqi Mohiuddin, and Matt Nawrocki (2017). *LabVIEW: Advanced Programming Techniques*. 2nd Edition. CRC Press, p. 520. ISBN: 9781315222097 (cited on page 153).

Black, David C., Jack Donovan, Bill Bunton, and Anna Keist (2009). *SystemC: From the Ground Up, Second Edition*. 2nd. Springer New York. ISBN: 978-0-387-69957-8 (cited on page 28).

Blessing, Sebastian, Kiko Fernandez-Reyes, Albert Mingkun Yang, Sophia Drossopoulou, and Tobias Wrigstad (2019). *Run, Actor, Run: Towards Cross-Actor Language Benchmarking*. In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2019. Athens, Greece: Association for Computing Machinery, pp. 41–50. ISBN: 9781450369824 (cited on page 103).

Blochwitz, Torsten, Martin Otter, Martin Arnold, Constanze Bausch, Christoph Clauß, Hilding Elmqvist, Andreas Junghanns, Jakob Mauss, Manuel Monteiro, Thomas Neidhold, Dietmar Neumerkel, Hans Olsson, Jörg-Volker Peetz, and Susann Wolf (2011). *The Functional Mockup Interface for Tool independent Exchange of Simulation Models*. In: *Proceedings of the 8th International Modelica Conference*. Ed. by Christoph Clauß. Linköping Electronic Conference Proceedings. Linköping University Press, pp. 105–114 (cited on page 28).

Blumofe, Robert D. and Charles E. Leiserson (1999). *Scheduling Multithreaded Computations By Work Stealing*. In: *Journal of the ACM* 46.5, pp. 720–748. ISSN: 0004-5411 (cited on page 155).

Bocchino, Robert, Timothy Canham, Garth Watney, Leonard Reder, and Jeffrey Levison (2018). *F Prime: an Open-Source Framework for Small-Scale Flight Software Systems*. In: *32nd Annual AIAA/USU Conference on Small Satellites* (cited on page 154).

Bojowald, Martin (Feb. 2017). *Now: the Physics of Time*. In: *Physics Today* 70.2, pp. 57–58. ISSN: 0031-9228 (cited on page 25).

Bouali, Amar (1998). *Xeve, an Esterel Verification Environment*. In: *Computer Aided Verification*. Ed. by Alan J. Hu and Moshe Y. Vardi. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 500–504. ISBN: 978-3-540-69339-0 (cited on page 31).

Boussinot, Frédéric and Robert de Simone (1991). *The Esterel Language*. In: *Proceedings of the IEEE* 79.9, pp. 1293–1304 (cited on page 30).

Box, George E.P. (1979). *Robustness in the Strategy of Scientific Model Building*. In: *Robustness in Statistics*. Ed. by Robert L. Launer and Graham N. Wilkinson. Academic Press, pp. 201–236. ISBN: 978-0-12-438150-6 (cited on page 2).

Boyapati, Chandrasekhar, Robert Lee, and Martin Rinard (2002). *Ownership Types for Safe Programming: Preventing Data Races and Deadlocks*. In: *SIGPLAN Notices* 37.11, pp. 211–230. ISSN: 0362-1340 (cited on page 59).

Brewer, Eric (2012). *CAP Twelve Years Later: How the "Rules" Have Changed*. In: *Computer* 45.2, pp. 23–29 (cited on page 84).

Broman, David, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter (2015). *Requirements for Hybrid Cosimulation Standards*. In: *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*. HSCC '15. Seattle, Washington: Association for Computing Machinery, pp. 179–188. ISBN: 9781450334334 (cited on page 26).

Brookes, Stephen, C. A. R. Hoare, and Andrew W. Roscoe (1984). *A Theory of Communicating Sequential Processes*. In: *Journal of the ACM* 31.3, pp. 560–599. ISSN: 0004-5411 (cited on page 40).

Brookes, Stephen and Peter W. O'Hearn (2016). *Concurrent Separation Logic*. In: *ACM SIGLOG News* 3.3, pp. 47–65 (cited on page 11).

Budhdev, Nishant, Mun Choon Chan, and Tulika Mitra (2018). *PR^3: Power Efficient and Low Latency Baseband Processing for LTE Femtocells*. In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pp. 2357–2365 (cited on page 147).

– (2020a). *Isoran: Isolation and Scaling for 5g Ranvia User-Level Data Plane Virtualization*. In: *CoRR* abs/2003.01841. arXiv: 2003.01841 (cited on pages 146, 147).

– (2020b). *Poster: IsoRAN: Isolation and Scaling for 5G RAN via User-Level Data Plane Virtualization*. In: *2020 IFIP Networking Conference (Networking)*, pp. 634–636 (cited on page 146).

Bünder, Hendrik (2019). *Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages*. In: *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*. MODELSWARD 2019. Prague, Czech Republic: SCITEPRESS - Science and Technology Publications, Lda, pp. 129–140. ISBN: 9789897583582 (cited on pages 85, 88).

Burns, Alan and Andy Wellings (2007). *Concurrent and real-time programming in Ada*. Cambridge University Press. ISBN: 9780521866972 (cited on page 154).

Callanan, Gareth and Flavius Gruian (2023). *Scalable Actor Networks with CAL*. In: *Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE '23. Hamburg, Germany: Association for Computing Machinery, pp. 169–179. ISBN: 9798400703188 (cited on page 155).

Casale-Brunet, Simone, Claudio Alberti, Marco Mattavelli, and Jorn W. Janneck (2013). *Turnus: A unified dataflow design space exploration framework for heterogeneous parallel systems*. In: *2013 Conference on Design and Architectures for Signal and Image Processing*, pp. 47–54. ISBN: 979-10-92279-01-6 (cited on page 137).

Cassandras, Christos G. (2016). *Smart Cities as Cyber-Physical Social Systems*. In: *Engineering* 2.2, pp. 156–158. ISSN: 2095-8099 (cited on page 1).

Castrillon, Jeronimo, Karol Desnos, Andrés Goens, and Christian Menard (Jan. 2023). *Dataflow Models of Computation for Programming Heterogeneous Multicores*. In: *Handbook of Computer Architecture*. Ed. by Anupam Chattopadhyay et al. Singapore: Springer Nature Singapore. ISBN: 978-981-15-6401-7 (cited on pages x, 5, 22, 135, 136).

Castrillon, Jeronimo and Rainer Leupers (2014). *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. Springer. ISBN: 978-3-319-00675-8 (cited on pages 5, 22, 135, 137, 140).

Castrillon, Jeronimo, Matthias Lieber, Sascha Klüppelholz, Marcus Völp, Nils Asmussen, Uwe Assmann, Franz Baader, Christel Baier, Gerhard Fettweis, Jochen Fröhlich, Andrés Goens, Sebastian Haas, Dirk Habich, Hermann Härtig, Mattis Hasler, Immo Huismann, Tomas Karnagel, Sven Karol, Akash Kumar, Wolfgang Lehner, Linda Leuschner, Siqi Ling, Steffen Märcker, Christian Menard, Johannes Mey, Wolfgang Nagel, Benedikt Nöthen, Rafael Peñaloza, Michael Raitza, Jörg Stiller, Annett Ungethüm, Axel Voigt, and Sascha Wunderlich (July 2018). *A Hardware/software Stack for Heterogeneous Systems*. In: *IEEE Transactions on Multi-Scale Computing Systems* 4.3, pp. 243–259. ISSN: 2332-7766 (cited on pages xi, 135).

Castrillon, Jeronimo, Weihua Sheng, and Rainer Leupers (2011). *Trends in Embedded Software Synthesis*. In: *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pp. 347–354 (cited on page 135).

Charousset, Dominik, Raphael Hiesgen, and Thomas C. Schmidt (2016). *Revisiting Actor Programming in C++*. In: *Computer Languages, Systems & Structures* 45, pp. 105–131. ISSN: 1477-8424 (cited on pages 15, 94).

Chung, Moo-Kyoung, Jun-Kyoung Kim, and Soojung Ryu (2014). *SimParallel: A High Performance Parallel SystemC Simulator Using Hierarchical Multi-threading*. In: *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1472–1475 (cited on page 29).

Church, Alonzo (1936). *An Unsolvable Problem of Elementary Number Theory*. In: *Journal of Symbolic Logic* 1.2, pp. 73–74 (cited on page 2).
Clebsch, Sylvan, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek (2017). *Orca: GC and Type System Co-Design for Actor Languages*. In: *Proc. ACM Program. Lang.* 1.OOPSLA (cited on page 15).
Cohen, Albert, Léonard Gérard, and Marc Pouzet (2012). *Programming Parallelism with Futures in Lustre*. In: *Proceedings of the Tenth ACM International Conference on Embedded Software*. EMSOFT '12. Tampere, Finland: Association for Computing Machinery, pp. 197–206. ISBN: 9781450314251 (cited on page 31).
Colaço, Jean-Louis, Bruno Pagano, and Marc Pouzet (2017). *SCADE 6: A formal language for embedded critical software development (invited paper)*. In: *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)* (cited on pages 31, 154).
Corbett, James C., Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford (2013). *Spanner: Google's Globally Distributed Database*. In: *ACM Transactions on Computer Systems* 31.3. ISSN: 0734-2071 (cited on pages 6, 52).
Coussy, Philippe and Adam Morawiec (2008). *High-Level Synthesis: From Algorithm to Digital Circuit*. 1st. Springer Publishing Company, Incorporated. ISBN: 1402085877 (cited on page 159).
Cox, David Richard (July 2005). *RITSim: Distributed SystemC Simulation*. PhD thesis (cited on page 29).
cppreference.com (2023a). *Date and Time Utilities*. URL: https://en.cppreference.com/mwiki/index.php?title=cpp/chrono&oldid=154276 (visited on July 21, 2023) (cited on page 27).
– (2023b). *std::shared_ptr*. URL: https://en.cppreference.com/w/cpp/memory/shared_ptr (visited on Dec. 5, 2023) (cited on page 91).
– (2023c). *std::unique_ptr*. URL: https://en.cppreference.com/w/cpp/memory/unique_ptr (visited on Dec. 5, 2023) (cited on page 91).
Dabney, James B. and Thomas L. Harman (2004). *Mastering Simulink*. Prentice Hall. ISBN: 978-0131424777 (cited on page 153).
Dahmann, Judith S., Richard M. Fujimoto, and Richard M. Weatherly (1997). *The Department of Defense High Level Architecture*. In: *Proceedings of the 29th Conference on Winter Simulation*. WSC '97. Atlanta, Georgia, USA: IEEE Computer Society, pp. 142–149. ISBN: 078034278X (cited on page 81).
Dawson, Bruce (Feb. 2012a). *Comparing Floating Point Numbers, 2012 Edition*. URL: https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition (visited on July 21, 2023) (cited on page 26).
– (Feb. 2012b). *Don't Store That in a Float*. URL: https://randomascii.wordpress.com/2012/02/13/dont-store-that-in-a-float (visited on July 21, 2023) (cited on page 26).
Denis, Xavier, Jacques-Henri Jourdan, and Claude Marché (2022). *Creusot: A Foundry for the Deductive Verification of Rust Programs*. In: *Formal Methods and Software Engineering*. Ed. by Adrian Riesco and Min Zhang. Cham: Springer International Publishing, pp. 90–105. ISBN: 978-3-031-17244-1 (cited on page 87).
Dennis, Jack B. (1974). *First Version of a Data Flow Procedure Language*. In: *Programming Symposium*. Ed. by B. Robinet. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 362–376. ISBN: 978-3-540-37819-8 (cited on page 20).
– (1986). *Data Flow Computation*. In: *Control Flow and Data Flow: Concepts of Distributed Programming*. Ed. by Manfred Broy. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 345–398. ISBN: 978-3-642-82921-5 (cited on page 20).
Derler, Patricia, Thomas Huining Feng, Edward A. Lee, Slobodan Matic, Hiren D. Patel, Yang Zhao, and Jia Zou (Mar. 2008). *PTIDES: A Programming Model for Distributed Real-Time Embedded Systems*. Tech. rep. UCB/EECS-2008-72. EECS Department, University of California, Berkeley (cited on pages 52, 82).
Derler, Patricia, Edward A. Lee, and Alberto Sangiovanni Vincentelli (2012). *Modeling Cyber-Physical Systems*. In: *Proceedings of the IEEE* 100.1, pp. 13–28 (cited on page 3).
Desai, Ankush, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey (2013). *P: Safe Asynchronous Event-Driven Programming*. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, pp. 321–332. ISBN: 9781450320146 (cited on page 15).
Desnos, Karol, Maxime Pelcat, Jean-François Nezan, Shuvra S. Bhattacharyya, and Slaheddine Aridhi (2013). *PiMM: Parameterized and Interfaced Dataflow Meta-Model for MPSoCs Runtime Reconfiguration*. In: *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 41–48 (cited on page 22).

Dick, Robert P., David L. Rhodes, and Wayne Wolf (1998). *TGFF: Task Graphs for Free*. In: *Proceedings of the Sixth International Workshop on Hardware/Software Codesign. (CODES/CASHE'98)*, pp. 97–101 (cited on page 141).
Dinning, Anne (1989). *A Survey of Synchronization Methods for Parallel Computers*. In: *Computer* 22.7, pp. 66–77 (cited on page 10).
Dmitrović, Slobodan (2023). *Smart Pointers*. In: *Modern C++ for Absolute Beginners: A Friendly Introduction to the C++ Programming Language and C++11 to C++23 Standards*. Berkeley, CA: Apress, pp. 211–215. ISBN: 978-1-4842-9274-7 (cited on pages 65, 91).
Dong, Zheng and Cong Liu (2017). *Analysis Techniques for Supporting Hard Real-Time Sporadic Gang Task Systems*. In: *2017 IEEE Real-Time Systems Symposium (RTSS)*, pp. 128–138 (cited on page 13).
[SW] Donovan, Peter and Marten Lohstroh, *Lingua Franca extension for Visual Studio Code*. LIC: BSD-2-Clause. URL: https://github.com/lf-lang/vscode-lingua-franca, (visited on Nov. 10, 2023) (cited on page 88).
Dragoni, Nicola, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina (2017). *Microservices: Yesterday, Today, and Tomorrow*. In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Cham: Springer International Publishing, pp. 195–216. ISBN: 978-3-319-67425-4 (cited on page 18).
Edelson, Daniel R. (1992). *Smart Pointers: They're Smart, But They're Not Pointers*. In: *Proceedings of the C++ Conference. Portland, OR, USA, August 1992*. USENIX Association, pp. 1–20 (cited on pages 65, 91).
Edwards, Stephen A. and Edward A. Lee (2003). *The Semantics and Execution of a Synchronous Block-Diagram Language*. In: *Science of Computer Programming* 48.1, pp. 21–42 (cited on page 30).
Eker, Johan and Jörn W. Janneck (2003). *CAL Language Report: Specification of the CAL Actor Language*. Tech. rep. UC Berkeley (cited on pages 22, 137).
Eker, Johan, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozref Ludvig, Stephem Neuendorffer, Sonia Sachs, and Yuhong Xiong (2003). *Taming Heterogeneity—the Ptolemy Approach*. In: *Proceedings of the IEEE* 91.1, pp. 127–144 (cited on page 22).
Erbas, Cagkan, Selin Cerav-Erbas, and Andy D. Pimentel (2006). *Multiobjective Optimization and Evolutionary Algorithms for the Application Mapping Problem in Multiprocessor System-On-Chip Design*. In: *IEEE Transactions on Evolutionary Computation* 10.3, pp. 358–374 (cited on page 145).
Ernst, Rolf, Leonie Ahrendts, and Kai-Björn Gemlau (2018). *System Level LET: Mastering Cause-Effect Chains in Distributed Systems*. In: *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, pp. 4084–4089 (cited on pages 33, 53).
Ertel, Sebastian (Dec. 2019). *Towards Implicit Parallel Programming for Systems*. PhD thesis. Dresden, Germany: TU Dresden (cited on pages 22, 155).
Eugster, Patrick Th., Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec (2003). *The Many Faces of Publish/subscribe*. In: *ACM Computing Surveys* 35.2, pp. 114–131. ISSN: 0360-0300 (cited on pages 18, 154).
Eysholdt, Moritz and Heiko Behrens (2010). *Xtext: Implement Your Language Faster than the Quick and Dirty Way*. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, pp. 307–309. ISBN: 9781450302401 (cited on page 85).
Feitelson, Dror G. and Larry Rudolph (1992). *Gang Scheduling Performance Benefits for Fine-Grain Synchronization*. In: *Journal of Parallel and Distributed Computing* 16.4, pp. 306–318. ISSN: 0743-7315 (cited on page 13).
Fersman, Elena, Paul Pettersson, and Wang Yi (2002). *Timed Automata with Asynchronous Processes: Schedulability and Decidability*. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Joost-Pieter Katoen and Perdita Stevens. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 67–82. ISBN: 978-3-540-46002-2 (cited on page 12).
Fidge, Colin J. (1988). *Timestamps in Message-Passing Systems That Preserve the Partial Ordering*. In: *Australian Computer Science Communications*, pp. 56–66 (cited on page 26).
Finn, Norman (2018). *Introduction To Time-Sensitive Networking*. In: *IEEE Communications Standards Magazine* 2.2, pp. 22–28 (cited on page 84).
Fishman, George S. (Sept. 2011). *Discrete-event Simulation: Modeling, Programming, and Analysis*. Springer New York. ISBN: 978-1-4419-2892-4 (cited on page 28).
Fortin, Félix-Antoine, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné (2012). *DEAP: Evolutionary Algorithms Made Easy*. In: *The Journal of Machine Learning Research* 13.1, pp. 2171–2175. ISSN: 1532-4435 (cited on page 145).
Fournier, Clément (Dec. 2021). *A Rust Backend for Lingua Franca*. MA thesis. TU Dresden (cited on page 87).
[SW] Fournier, Clément and Johannes Hayeß, *reactor-rs*. LIC: MIT. URL: https://github.com/lf-lang/reactor-ts, (visited on Nov. 10, 2023) (cited on page 87).

Friedmann, Mattern (1988). *Virtual Time and Global States of Distributed Systems*. In: *Proceedings of the 10th International Workshop on Parallel and Distributed Algorithms, October 1988*. North-Holland (cited on page 26).
Fuhrmann, Hauke and Reinhard von Hanxleden (2010). *Taming Graphical Modeling*. In: *Model Driven Engineering Languages and Systems*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 196–210. ISBN: 978-3-642-16145-2 (cited on page 87).
Geilen, Marc and Twan Basten (2003). *Requirements on the Execution of Kahn Process Networks*. In: *Programming Languages and Systems*. Ed. by Pierpaolo Degano. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 319–334. ISBN: 978-3-540-36575-4 (cited on page 20).
Gemlau, Kai-Björn, Leonie Köhler, Rolf Ernst, and Sophie Quinton (2021). *System-Level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-Time Automotive Software*. In: *ACM Transactions on Cyber-Physical Systems* 5.2. ISSN: 2378-962X (cited on pages 13, 33, 53).
Goens, Andrés (May 2021). *Improving Model-Based Software Synthesis: A Focus on Mathematical Structures*. PhD thesis. TU Dresden (cited on pages 19, 21, 136, 144).
Goens, Andrés, Robert Khasanov, Jeronimo Castrillon, Simon Polstra, and Andy Pimentel (2016). *Why Comparing System-Level MPSoC Mapping Approaches is Difficult: A Case Study*. In: *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pp. 281–288 (cited on pages 137, 145).
Goens, Andrés, Robert Khasanov, Marcus Hähnel, Till Smejkal, Hermann Härtig, and Jeronimo Castrillon (June 2017). *TETRiS: a Multi-Application Run-Time System for Predictable Execution of Static Mappings*. In: *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES'17)*. SCOPES '17. Sankt Goar, Germany: ACM, pp. 11–20. ISBN: 978-1-4503-5039-6 (cited on pages 145, 148).
Goens, Andrés, Christian Menard, and Jeronimo Castrillon (Sept. 2018). *On the Representation of Mappings to Multicores*. In: *Proceedings of the IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-18)*. Vietnam National University, Hanoi, Vietnam, pp. 184–191. ISBN: 978-1-5386-6689-0 (cited on pages xi, 144).
– (July 2019). *On Compact Mappings for Multicore Systems*. In: *Proceedings of the IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS)*. Ed. by D. Pnevmatikatos, M. Pelcat, and M. Jung. Vol. 11733. IEEE. Pythagorion, Greece: Springer, Cham, pp. 325–335. ISBN: 978-3-030-27561-7 (cited on page xii).
Goens, Andrés, Sergio Siccha, and Jeronimo Castrillon (July 2017). *Symmetry in Software Synthesis*. In: *ACM Transactions on Architecture and Code Optimization (TACO)*, 14.2, 20:1–20:26. ISSN: 1544-3566 (cited on pages 144, 145, 148).
[SW] Google, *Catapult*. URL: https://chromium.googlesource.com/catapult (visited on Dec. 18, 2023) (cited on page 144).
Gu, Rui, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu (2015). *What Change History Tells Us about Thread Synchronization*. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, pp. 426–438. ISBN: 9781450336758 (cited on page 37).
Haas, Sebastian, Tobias Seifert, Benedikt Nöthen, Stefan Scholze, Sebastian Höppner, Andreas Dixius, Esther Pérez Adeva, Thomas Augustin, Friedrich Pauls, Sadia Moriam, Mattis Hasler, Erik Fischer, Yong Chen, Emil Matúš, Georg Ellguth, Stephan Hartmann, Stefan Schiefer, Love Cederström, Dennis Walter, Stephan Henker, Stefan Hänzsche, Johannes Uhlig, Holger Eisenreich, Stefan Weithoffer, Norbert Wehn, René Schüffny, Christian Mayr, and Gerhard Fettweis (2017). *A Heterogeneous SDR MPSoC in 28 Nm CMOS for Low-Latency Wireless Applications*. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC '17. Austin, TX, USA: Association for Computing Machinery. ISBN: 9781450349277 (cited on page 135).
Hagen, George and Cesare Tinelli (2008). *Scaling Up the Formal Verification of Lustre Programs with SMT-Based Techniques*. In: *2008 Formal Methods in Computer-Aided Design* (cited on page 31).
Halbwachs, Nicolas (1993). *Synchronous Programming of Reactive Systems*. USA: Kluwer Academic Publishers. ISBN: 0792393112 (cited on page 29).
Halbwachs, Nicolas, Paul Caspi, Pascal Raymond, and Daniel Pilaud (1991). *The Synchronous Data Flow Programming Language Lustre*. In: *Proceedings of the IEEE* 79.9, pp. 1305–1320 (cited on page 30).
Hameed, Fazal, Christian Menard, and Jeronimo Castrillon (Oct. 2017). *Efficient STT-RAM Last-Level-Cache Architecture to replace DRAM Cache*. In: *Proceedings of the International Symposium on Memory Systems (MemSys'17)*. MEMSYS '17. Alexandria, Virginia: ACM, pp. 141–151. ISBN: 978-1-4503-5335-9 (cited on page xii).
Hanxleden, Reinhard von, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien (2014). *SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications: HW/SW-Synthesis for a Conservative Extension of Synchronous Statecharts*. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: Association for Computing Machinery, pp. 372–383. ISBN: 9781450327848 (cited on page 31).

Hanxleden, Reinhard von, Edward A. Lee, Hauke Fuhrmann, Alexander Schulz-Rosengarten, Sören Domrös, Marten Lohstroh, Soroush Bateni, and Christian Menard (2022). *Pragmatics Twelve Years Later: A Report on Lingua Franca*. In: *Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering*. Springer Nature Switzerland, pp. 60–89 (cited on pages x, 41, 87).

Hanxleden, Reinhard von, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'brien, and Partha Roop (2014). *Sequentially Constructive Concurrency-A Conservative Extension of the Synchronous Model of Computation*. In: *ACM Transactions on Embedded Computing Systems* 13.4s. ISSN: 1539-9087 (cited on page 31).

Harel, David (1987). *Statecharts: a Visual Formalism for Complex Systems*. In: *Science of Computer Programming* 8.3, pp. 231–274. ISSN: 0167-6423 (cited on page 40).

Hayeß, Johannes (Mar. 2023). *Verifying the Rust Runtime of Lingua Franca*. MA thesis. TU Dresden (cited on page 87).

Hedden, Brandon and Xinghui Zhao (2018). *A Comprehensive Study on Bugs in Actor Systems*. In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018. Eugene, OR, USA: Association for Computing Machinery. ISBN: 9781450365109 (cited on page 17).

Hempel, Gerald, Andrés Goens, Jeronimo Castrillon, Josefine Asmus, and Ivo F. Sbalzarini (2017). *Robust Mapping of Process Networks to Many-Core Systems Using Bio-Inspired Design Centering*. In: *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '17. Sankt Goar, Germany: Association for Computing Machinery, pp. 21–30. ISBN: 9781450350396 (cited on page 145).

Hennig, Julien, Hermann von Hasseln, Hassan Mohammad, Stefan Resmerita, Stefan Lukesch, and Andreas Naderlinger (2016). *Towards Parallelizing Legacy Embedded Control Software Using the LET Programming Paradigm*. In: *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (cited on page 33).

Henzinger, Thomas A., Benjamin Horowitz, and Christoph M. Kirsch (2003). *Giotto: A Time-Triggered Language for Embedded Programming*. In: *Proceedings of the IEEE* 91.1, pp. 84–99 (cited on pages 33, 154).

Henzinger, Thomas A. and Joseph Sifakis (2006). *The Embedded Systems Design Challenge*. In: *FM 2006: Formal Methods*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-37216-5 (cited on pages 3, 6).

Heulot, Julien, Jani Boutellier, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi (2013). *Applying the Adaptive Hybrid Flow-Shop Scheduling Method to Schedule a 3GPP LTE Physical Layer Algorithm onto Many-core Digital Signal Brocessors*. In: *2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013)*, pp. 123–129 (cited on page 146).

Heulot, Julien, Maxime Pelcat, Karol Desnos, Jean-François Nezan, and Slaheddine Aridhi (2014). *SPIDER: A Synchronous Parameterized and Interfaced Dataflow-based RTOS for Multicore DSPs*. In: *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pp. 167–171 (cited on page 137).

Hewitt, Carl (1977). *Viewing Control Structures as Patterns of Passing Messages*. In: *Artificial Intelligence* 8.3, pp. 323–364 (cited on page 14).

Hewitt, Carl, Peter Bishop, and Richard Steiger (1973). *A Universal Modular ACTOR Formalism for Artificial Intelligence*. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., pp. 235–245 (cited on pages 14, 153).

Hirzel, Martin, Henrique Andrade, Buğra Gedik, Vibhore Kumar, Mark P. Mendell, Howard Nasgaard, Robert Soulé, and Kun-Lung Wu (2009). *SPL Stream Processing Language Specification*. Tech. rep. IBM (cited on page 22).

Hirzel, Martin, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Akrivi Vlachou (2018). *Stream Processing Languages in the Big Data Era*. In: *SIGMOD Record* 47.2, pp. 29–40. ISSN: 0163-5808 (cited on page 22).

Hirzel, Martin, Scott Schneider, and Buğra Gedik (2017). *SPL: an Extensible Language for Distributed Stream Processing*. In: *ACM Transactions on Programming Languages and Systems* 39.1. ISSN: 0164-0925 (cited on page 22).

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall. ISBN: 978-0131532892 (cited on page 40).

Hong, Shin and Moonzoo Kim (2015). *A Survey of Race Bug Detection Techniques for Multithreaded Programmes*. In: *Software Testing, Verification and Reliability* 25.3, pp. 191–217 (cited on page 11).

Huang, Kai, Iuliana Bacivarov, Fabian Hugelshofer, and Lothar Thiele (2008). *Scalably Distributed SystemC Simulation for Embedded Applications*. In: *2008 International Symposium on Industrial Embedded Systems*, pp. 271–274 (cited on page 29).

Hui, John and Stephen A. Edwards (2022). *The Sparse Synchronous Model on Real Hardware*. In: *ACM Transactions on Embedded Computing Systems*. Just Accepted. ISSN: 1539-9087 (cited on page 32).

Hui, John and Stephen A. Edwards (2023). *Towards Sparse Synchronous Programming in Lua*. In: *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. CPS-IoT Week '23. San Antonio, TX, USA: Association for Computing Machinery, pp. 361–366. ISBN: 9798400700491 (cited on page 32).

IEEE (2019). *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. IEEE Standard 1588-2019 (cited on page 84).

IEEE Computer Society (2019). *IEEE Standard for Software-Hardware Interface for Multi-Many-Core*. In: *IEEE 2804-2019* (cited on page 141).

IEEE Computer Society and The Open Group (2018). *IEEE Standard for Software-Hardware Interface for Multi-Many-Core*. In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (cited on page 26).

Imam, Shams M. and Vivek Sarkar (2014). *Savina – An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries*. In: *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*. AGERE! '14. Portland, Oregon, USA: Association for Computing Machinery, pp. 67–80. ISBN: 9781450321891 (cited on pages 94, 103, 109).

Jagadeesan, Lalita Jategaonkar, Carlos Puchol, and James E. Von Olnhausen (1995). *Safety Property Verification of Esterel Programs and Applications to Telecommunications Software*. In: *Computer Aided Verification*. Ed. by Pierre Wolper. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 127–140. ISBN: 978-3-540-49413-3 (cited on page 31).

Jazdi, Nasser (2014). *Cyber Physical Systems in the Context of Industry 4.0*. In: *2014 IEEE International Conference on Automation, Quality and Testing, Robotics* (cited on page 1).

Kahn, Gilles (1974). *The Semantics of a Simple Language for Parallel Programming*. In: *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*. Ed. by Jack L. Rosenfeld. North-Holland Publishing Co., pp. 471–475 (cited on pages 19, 153).

Kahn, Gilles and David B. MacQueen (1977). *Coroutines and Networks of Parallel Processes*. In: *Information Processing*. Ed. by B. Gilchrist. North-Holland Publishing Co., pp. 993–998 (cited on page 19).

Kalray Inc (2023). *MPPA DPU Architecture*. URL: https://www.kalrayinc.com/products/mppa-technology/ (visited on July 13, 2023) (cited on page 135).

Karnaugh, Maurice (1953). *The Map Method for Synthesis of Combinational Logic Circuits*. In: *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics* 72.5, pp. 593–599 (cited on page 6).

Keinert, Joachim, Martin Streubühr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jürgen Teich, and Michael Meredith (2009). *Systemcodesigner—An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications*. In: *ACM Transactions on Design Automation of Electronic Systems* 14.1. ISSN: 1084-4309 (cited on page 137).

Khasanov, Robert and Jeronimo Castrillon (2020). *Energy-efficient Runtime Resource Management for Adaptable Multi-application Mapping*. In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 909–914 (cited on pages 145, 148, 149).

Khasanov, Robert, Andrés Goens, and Jeronimo Castrillon (2018). *Implicit Data-Parallelism in Kahn Process Networks: Bridging the MacQueen Gap*. In: *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*. PARMA-DITAM '18. Manchester, United Kingdom: Association for Computing Machinery, pp. 20–25. ISBN: 9781450364447 (cited on page 19).

Khasanov, Robert, Julian Robledo, Christian Menard, Andrés Goens, and Jeronimo Castrillon (Sept. 2021). *Domain-Specific Hybrid Mapping for Energy-Efficient Baseband Processing in Wireless Networks*. In: *ACM Transactions on Embedded Computing Systems (TECS). Special issue of the International Conference on Compilers, Architecture, and Synthesis of Embedded Systems (CASES)* 20.5s. ISSN: 1539-9087 (cited on pages x, 146, 147, 149, 150).

Kim, Ki Hyung, Yeong Rak Seong, Tag Gon Kim, and Kyu Ho Park (1997). *Ordering of Simultaneous Events in Distributed DEVS Simulation*. In: *Simulation Practice and Theory* 5.3, pp. 253–268. ISSN: 0928-4869 (cited on page 27).

Kirsch, Christoph M. and Ana Sokolova (2012). *The Logical Execution Time Paradigm*. In: *Advances in Real-Time Systems*. Ed. by Samarjit Chakraborty and Jörg Eberspächer. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 103–120. ISBN: 978-3-642-24349-3 (cited on pages 32, 153).

Klabnik, Steve and Carol Nichols (2022). *The Rust Programming Language*. 2nd. No Starch Press. ISBN: 9781718503106 (cited on page 59).

Kleene, Stephen C. (1936). *General Recursive Functions of Natural Numbers*. In: *Mathematische Annalen* 112, pp. 727–742 (cited on page 2).

Kligerman, Eugene and Alexander D. Stoyenko (1986). *Real-Time Euclid: a Language for Reliable Real-Time Systems*. In: *IEEE Transactions on Software Engineering* SE-12.9, pp. 941–949 (cited on page 154).

Köhler, Leonie, Phil Hertha, Matthias Beckert, Alex Bendrick, and Rolf Ernst (2023). *Robust Cause-Effect Chains With Bounded Execution Time and System-Level Logical Execution Time*. In: *ACM Trans. Embed. Comput. Syst.* 22.3. ISSN: 1539-9087 (cited on page 33).

Kopetz, Hermann (Apr. 2011). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer New York, NY. ISBN: 978-1-4419-8236-0 (cited on page 3).

Kopetz, Hermann and Günther Bauer (2003). *The Time-Triggered Architecture*. In: *Proceedings of the IEEE* 91.1, pp. 112–126 (cited on page 33).

Koubaa, Anis, ed. (Feb. 2016). *Robot Operating System (ROS)—The Complete Reference (Volume 1)*. Springer Cham. ISBN: 978-3-319-26054-9 (cited on pages 37, 154).

Kounev, Samuel, Nikolas Herbst, Cristina L. Abad, Alexandru Iosup, Ian Foster, Prashant Shenoy, Omer Rana, and Andrew A. Chien (2023). *Serverless Computing: What It Is, and What It Is Not?* In: *Commun. ACM* 66.9, pp. 80–92. ISSN: 0001-0782 (cited on page 1).

Krook, Robert, John Hui, Bo Joel Svensson, Stephen A. Edwards, and Koen Claessen (2022). *Creating a Language for Writing Real-Time Applications for the Internet of Things*. In: *2022 20th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)* (cited on pages 32, 154).

Kuhl, Frederick, Richard M. Weatherly, and Judith S. Dahmann (1999). *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. USA: Prentice Hall PTR. ISBN: 0130225118 (cited on page 81).

Kwok, Yu-Kwong and Ishfaq Ahmad (1999). *Static Scheduling Algorithms for Allocating Directed Task Graphs To Multiprocessors*. In: *ACM Computing Surveys* 31.4, pp. 406–471. ISSN: 0360-0300 (cited on pages 14, 99).

Lakshmanan, Karthik, Shinpei Kato, and Ragunathan Rajkumar (2010). *Scheduling Parallel Real-Time Tasks on Multi-core Processors*. In: *2010 31st IEEE Real-Time Systems Symposium*, pp. 259–268 (cited on page 13).

Lall, Sanjay, Calin Cascaval, Martin Izzard, and Tammo Spalink (2023). *Logical Synchrony and the Bittide Mechanism*. In: *CoRR* abs/2308.00144. arXiv: 2308.00144(cited on pages 154, 158).

Lamport, Leslie (1978). *Time, Clocks, and the Ordering of Events in a Distributed System*. In: *Commun. ACM* 21.7, pp. 558–565. ISSN: 0001-0782 (cited on page 25).

Lanese, Ivan, Naoki Nishida, Adrián Palacios, and Germán Vidal (2018). *CauDEr: A Causal-Consistent Reversible Debugger for Erlang*. In: *Functional and Logic Programming*. Ed. by John P. Gallagher and Martin Sulzmann. Cham: Springer International Publishing, pp. 247–263. ISBN: 978-3-319-90686-7 (cited on page 17).

Laskey, Kathryn B. and Kenneth Laskey (2009). *Service Oriented Architecture*. In: *WIREs Computational Statistics* 1.1, pp. 101–105 (cited on pages 18, 154).

Le Guernic, Paul, Thierry Gautier, Michel Le Borgne, and Claude Le Maire (1991). *Programming Real-Time Applications With Signal*. In: *Proceedings of the IEEE* 79.9, pp. 1321–1336 (cited on page 30).

Lee, Edward A. (2006). *The Problem With Threads*. In: *Computer* 39.5, pp. 33–42 (cited on page 10).

– (2008). *Cyber Physical Systems: Design Challenges*. In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 363–369 (cited on page 3).

– (2009). *Computing Needs Time*. In: *Communications of the ACM* 52.5, pp. 70–79. ISSN: 0001-0782 (cited on pages 1, 24).

– (Oct. 2018). *Plato and the Nerd—The Creative Partnership of Humans and Technology*. The MIT Press. ISBN: 9780262536424 (cited on page 2).

– (2019). *Freedom From Choice and the Power of Models: In Honor of Alberto Sangiovanni-Vincentelli*. In: *Proceedings of the 2019 International Symposium on Physical Design*. ISPD '19. San Francisco, CA, USA: Association for Computing Machinery, p. 126. ISBN: 9781450362535 (cited on page 2).

– (July 2021). *Determinism*. In: *ACM Transactions on Embedded Computing Systems (TECS)* 20.5, pp. 1–34 (cited on pages 3, 4).

Lee, Edward A., Ravi Akella, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard (2023). *Consistency Vs. Availability in Distributed Cyber-Physical Systems*. In: *ACM Transactions on Embedded Computing Systems* 22.5s. ISSN: 1539-9087 (cited on pages x, 58, 85).

Lee, Edward A., Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard (2021). *Quantifying and Generalizing the CAP Theorem* (cited on pages xi, 85).

– (2023). *Trading Off Consistency and Availability in Tiered Heterogeneous Distributed Systems*. In: *Intelligent Computing* 2 (cited on pages x, 58, 85).

Lee, Edward A. and Marten Lohstroh (2022). *Generalizing Logical Execution Time*. In: *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*. Ed. by Jean-François Raskin, Krishnendu Chatterjee, Laurent Doyen, and Rupak Majumdar. Cham: Springer Nature Switzerland, pp. 160–181. ISBN: 978-3-031-22337-2 (cited on pages 84, 126).

Lee, Edward A. and Eleftherios Matsikoudis (Sept. 2009). *The Semantics of Dataflow With Firing*. In: *From Semantics to Computer Science: Essays in Honour of Gilles Kahn* (cited on pages 19, 21).

Lee, Edward A. and David G. Messerschmitt (1987). *Synchronous Data Flow*. In: *Proceedings of the IEEE* 75.9, pp. 1235–1245 (cited on pages 21, 153).
Lee, Edward A. and Thomas M. Parks (1995). *Dataflow Process Networks*. In: *Proceedings of the IEEE* 83.5, pp. 773–801 (cited on page 21).
Lee, Edward A. and Sanjit A. Seshia (2016). *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. 2nd. The MIT Press. ISBN: 0262533812 (cited on pages 1, 30).
Lee, Edward A. and Haiyang Zheng (2007). *Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems*. In: *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*. EMSOFT '07. Salzburg, Austria: Association for Computing Machinery, pp. 114–123. ISBN: 9781595938251 (cited on page 29).
Leupers, Rainer and Jeronimo Castrillon (2010). *MPSoC Programming Using the MAPS Compiler*. In: *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 897–902 (cited on page 137).
Li, Xin (2007). *The Kiel Esterel Processor: A Multi-threaded Reactive Processor*. PhD thesis. Christian-Albrechts Universität Kiel (cited on page 31).
Li, Xin and Reinhard von Hanxleden (2012). *Multithreaded Reactive Programming—The Kiel Esterel Processor*. In: *IEEE Transactions on Computers* 61.3, pp. 337–349 (cited on page 31).
Lieberman, Henry (1987). *Concurrent Object-Oriented Programming in Act 1*. In: *Object-Oriented Concurrent Programming*. Cambridge, MA, USA: MIT Press, pp. 9–36. ISBN: 0262240262 (cited on page 15).
Liu, C. L. and James W. Layland (1973). *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. In: *Journal of the ACM* 20.1, pp. 46–61. ISSN: 0004-5411 (cited on page 11).
Liu, Tongping, Charlie Curtsinger, and Emery D. Berger (2011). *Dthreads: Efficient Deterministic Multithreading*. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: Association for Computing Machinery, pp. 327–336. ISBN: 9781450309776 (cited on page 11).
Liu, Ziheng, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song (2021). *Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems*. In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, pp. 616–629. ISBN: 9781450383172 (cited on page 11).
Lohstroh, Marten (Dec. 2020). *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*. PhD thesis (cited on pages 1, 7, 41, 44, 100, 103, 153).
Lohstroh, Marten, Soroush Bateni, Christian Menard, Alexander Schulz-Rosengarten, Jeronimo Castrillon, and Edward A. Lee (2023). *Deterministic Coordination Across Multiple Timelines*. In: *ACM Transactions on Embedded Computing Systems*. ISSN: 1539-9087 (cited on pages x, 60).
[SW] Lohstroh, Marten, Hokeun Kim, Matt Weber, and Byeong-gil Jun, *reactor-ts*. URL: https://github.com/lf-lang/reactor-ts, (visited on Nov. 10, 2023) (cited on page 87).
Lohstroh, Marten and Edward A. Lee (2019a). *Deterministic Actors*. In: *2019 Forum for Specification and Design Languages (FDL)* (cited on pages 19, 41).
– (2019b). *Work-in-Progress: Real-Time Reactors in C*. In: *2019 IEEE Real-Time Systems Symposium (RTSS)*, pp. 572–575 (cited on page 58).
[SW] Lohstroh, Marten, Edward A. Lee, Soroush Bateni, Christian Menard, Peter Donovan, Clément Fournier, Hou Seng Wong, Alexander Schulz-Rosengarten, Erling Rennemo Jellum, Hokeun Kim, Matt Weber, Shaokai Lin, and Anirudh Rengarajan, *Lingua Franca*. LIC: BSD-2-Clause. URL: https://github.com/lf-lang/lingua-franca (visited on Nov. 10, 2023) (cited on page 87).
Lohstroh, Marten, Edward A. Lee, Stephen A. Edwards, and David Broman (2023). *Logical Time for Reactive Software*. In: *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. CPS-IoT Week '23. San Antonio, TX, USA: Association for Computing Machinery, pp. 313–318. ISBN: 9798400700491 (cited on pages 25, 26, 30).
Lohstroh, Marten, Christian Menard, Soroush Bateni, and Edward A. Lee (2021). *Toward a Lingua Franca for Deterministic Concurrent Systems*. In: *ACM Transactions on Embedded Computing Systems* 20.4, pp. 1–27 (cited on pages x, 60).
Lohstroh, Marten, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A. Lee (Sept. 2020). *A Language for Deterministic Coordination Across Multiple Timelines*. In: *2020 Forum for Specification and Design Languages (FDL)*. Kiel, Germany, pp. 1–8 (cited on pages xi, 60).
Lohstroh, Marten, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli (2019). *Reactors: A Deterministic Model for Composable Reactive Systems*. In: *Cyber Physical Systems. Model-Based Design: 9th International Workshop, CyPhy 2019, and 15th International Workshop, WESE 2019, New York City, NY, USA, October 17-18, 2019, Revised Selected Papers*. New York City, NY, USA: Springer-Verlag, pp. 59–85. ISBN: 978-3-030-41130-5 (cited on pages 1, 7, 41, 44, 103, 153).

Lohstroh, Marten, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee (2019). *Actors Revisited for Time-Critical Systems*. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC '19. Las Vegas, NV, USA: Association for Computing Machinery. ISBN: 9781450367257 (cited on page 41).

[SW] Lohstroh, Marten, Alexander Schulz-Rosengarten, Soroush Bateni, Christian Menard, and Peter Donovan, *Epoch IDE for Lingua Franca*. LIC: BSD-2-Clause. URL: https://github.com/lf-lang/epoch (visited on Nov. 10, 2023) (cited on page 88).

Lowe-Power, Jason, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian (July 2020). *The Gem5 Simulator: Version 20.0+*. In: *arXiv preprint arXiv:2007.03152* (cited on pages xi, 28).

Macenski, Steven, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall (May 2022). *Robot Operating System 2: Design, Architecture, and Uses in the Wild*. In: *Science Robotics* 7.66. ISSN: 2470-9476 (cited on page 37).

Maillet-Contoz, Laurent and Frank Ghenassia (2005). *Transaction Level Modeling*. In: *Transaction Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Ed. by Frank Ghenassia. Boston, MA: Springer US, pp. 23–55. ISBN: 978-0-387-26233-8 (cited on page 28).

Maler, Oded, Zohar Manna, and Amir Pnueli (1992). *From Timed to Hybrid Systems*. In: *Real-Time: Theory in Practice*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 447–484. ISBN: 978-3-540-47218-6 (cited on page 27).

Manolache, Sorin, Petru Eles, and Zebo Peng (2008). *Task Mapping and Priority Assignment for Soft Real-Time Applications Under Deadline Miss Ratio Constraints*. In: *ACM Transactions on Embedded Computing Systems* 7.2. ISSN: 1539-9087 (cited on page 145).

Marwedel, Peter (2021). *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. 4th ed. Springer Cham. ISBN: 978-3-030-60909-2 (cited on page 3).

Mathur, Aman Shankar, Burcu Kulahcioglu Ozkan, and Rupak Majumdar (2018). *IDeA: An Immersive Debugger for Actors*. In: *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang*. Erlang 2018. St. Louis, MO, USA: Association for Computing Machinery. ISBN: 9781450358248 (cited on page 17).

Matloff, Norm (Feb. 2008). *Introduction To Discrete-Event Simulation and the Simpy Language*. In: (cited on page 28).

Mayr, Christian, Sebastian Hoeppner, and Steve Furber (2019). *SpiNNaker 2: A 10 Million Core Processor System for Brain Simulation and Machine Learning*. arXiv: 1911.02385 [cs.ET] (cited on page 135).

Mazumder, Sudip K., Abhijit Kulkarni, Subham Sahoo, Frede Blaabjerg, H. Alan Mantooth, Juan Carlos Balda, Yue Zhao, Jorge A. Ramos-Ruiz, Prasad N. Enjeti, P. R. Kumar, Le Xie, Johan H. Enslin, Burak Ozpineci, Anuradha Annaswamy, Herbert L. Ginn, Feng Qiu, Jianzhe Liu, Besma Smida, Colin Ogilvie, Juan Ospina, Charalambos Konstantinou, Mark Stanovich, Karl Schoder, Michael Steurer, Tuyen Vu, Lina He, and Eduardo Pilo de la Fuente (2021). *A Review of Current Research Trends in Power-Electronic Innovations in Cyber-Physical Systems*. In: *IEEE Journal of Emerging and Selected Topics in Power Electronics* 9.5, pp. 5146–5163 (cited on page 1).

McFarland, Michael C., Alice C. Parker, and Raul Camposano (1990). *The High-Level Synthesis of Digital Systems*. In: *Proceedings of the IEEE* 78.2, pp. 301–318 (cited on page 159).

Meijer, Erik (2010). *Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues*. In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, Maryland: Association for Computing Machinery. ISBN: 9781450305167 (cited on page 154).

[SW] Menard, Christian, *DEAR: Discrete Events for Adaptive AUTOSAR*. LIC: ISC. URL: https://github.com/tud-ccc/dear (visited on Nov. 10, 2023) (cited on page 50).

Menard, Christian, Andrés Goens, and Jeronimo Castrillon (Nov. 2016). *High-Level NoC Model for MPSoC Compilers*. In: *Proceedings of the IEEE Nordic Circuits and Systems Conference (NORCAS'16)*. NORCAS. Copenhagen, Denmark (cited on pages xii, 140).

Menard, Christian, Andrés Goens, Gerald Hempel, Robert Khasanov, Julian Robledo, Felix Teweleitt, and Jeronimo Castrillon (Jan. 2021). *Mocasin—Rapid Prototyping of Rapid Prototyping Tools: A Framework for Exploring New Approaches in Mapping Software to Heterogeneous Multi-cores*. In: *Proceedings of the 2021 Drone Systems Engineering*

and Rapid Simulation and Performance Evaluation: Methods and Tools, co-located with 16th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC). DroneSE and RAPIDO '21. Budapest, Hungary: Association for Computing Machinery, pp. 66–73. ISBN: 9781450389525 (cited on pages x, 136).

Menard, Christian, Andrés Goens, Marten Lohstroh, and Jeronimo Castrillon (Mar. 2020). *Achieving Determinism in Adaptive AUTOSAR*. In: *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE)*. DATE '20. Grenoble, France: IEEE, pp. 822–827. ISBN: 978-3-9819263-4-7 (cited on pages xi, 34, 50, 54, 58).

[SW] Menard, Christian, Andrés Goens, Robert Khasanov Felix Teweleitt, Julian Robledo, and Gerald Hempel, *Mocasin*. LIC: ISC. URL: https://github.com/tud-ccc/mocasin (visited on Dec. 19, 2023) (cited on page 137).

Menard, Christian, Matthias Jung, Jeronimo Castrillon, and Norbert Wehn (July 2017). *System Simulation with gem5 and SystemC: The Keystone for Full Interoperability*. In: *Proceedings of the IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS)*. IEEE. Pythagorion, Greece, pp. 62–69. ISBN: 978-1-5386-3437-0 (cited on pages xii, 28).

Menard, Christian, Marten Lohstroh, Soroush Bateni, Matthew Chorlian, Arthur Deng, Peter Donovan, Clément Fournier, Shaokai Lin, Felix Suchert, Tassilo Tanneberger, Hokeun Kim, Jeronimo Castrillon, and Edward A. Lee (2023). *High-Performance Deterministic Concurrency Using Lingua Franca*. In: *ACM Transactions on Architecture and Code Optimization*. Just Accepted. ISSN: 1544-3566 (cited on pages x, 15, 41, 60, 94).

[SW] Menard, Christian, Julian Robledo, and Robert Khasanov, *Fivegsim: A Simualtor for 5G Baseband Applications Based on Mocasin*. LIC: ISC. URL: https://github.com/tud-ccc/mocasin-fivegsim (visited on Dec. 19, 2023) (cited on page 147).

[SW] Menard, Christian, Bateni Soroush, Peter Donovan, Johannes Hayeß, Wonseo Choi, Marten Lohstro, and Matt Chorlian, *Lingua Franca Benchmarks*. URL: https://github.com/lf-lang/benchmarks-lingua-franca (visited on Nov. 13, 2023) (cited on page 105).

[SW] Menard, Christian and Tassilo Tanneberger, *reactor-cpp*. LIC: ISC. URL: https://github.com/lf-lang/reactor-cpp (visited on Nov. 10, 2023) (cited on pages 87, 89).

Merrifield, Timothy, Joseph Devietti, and Jakob Eriksson (2015). *High-Performance Determinism with Total Store Order Consistency*. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: Association for Computing Machinery. ISBN: 9781450332385 (cited on page 11).

Merrifield, Timothy, Sepideh Roghanchi, Joseph Devietti, and Jakob Eriksson (2019). *Lazy Determinism for Faster Deterministic Multithreading*. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, pp. 879–891. ISBN: 9781450362405 (cited on page 11).

Milner, Robin (1999). *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press. ISBN: 9780521658690 (cited on page 40).

MinnowBoard.org Foundation (2023). *MinnowBoard Turbot Dual Ethernet Family Technical Specs*. URL: https://www.minnowboard.org/minnowboard-turbot-dual-e/technical-specs/ (visited on Aug. 11, 2023) (cited on page 55).

Mok, Aloysius K. (1983). *Fundamental Design Problems of Distributed Systems for the Hard-real-time Environment*. PhD thesis. Massachusetts Institute of Technology. Department of Electrical Engineering and Computer Science (cited on page 12).

Mok, Aloysius K. and Deji Chen (1997). *A Multiframe Model for Real-Time Tasks*. In: *IEEE Transactions on Software Engineering* 23.10, pp. 635–645 (cited on page 12).

Molnar, Ingo (2023). *Design of the CFS scheduler*. URL: http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt (visited on Dec. 18, 2023) (cited on page 145).

Moore, Gordon (Apr. 1965). *Cramming More Components Onto Integrated Circuits*. In: *Electronics Magazine* (cited on pages 5, 135).

Moritz, Philipp, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica (2018). *Ray: A Distributed Framework for Emerging AI Applications*. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, pp. 561–577. ISBN: 9781931971478 (cited on page 15).

Mulazzani, Marco (1985). *Reliability Versus Safety*. In: *IFAC Proceedings Volumes* 18.12. 4th IFAC Workshop on Safety of Computer Control Systems 1985 (SAFECOMP'85): Achieving Safe Real Time Computer Systems, Como, Italy, 1-3 October 1985, pp. 141–146. ISSN: 1474-6670 (cited on page 3).

Murillo, Luis Gabriel, Simon Wawroschek, Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid (Mar. 2014). *Automatic Detection of Concurrency Bugs through Event Ordering Constraints*. In: *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. Dresden, Germany (cited on page 11).

Murray, Derek G., Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi (2013). *Naiad: A Timely Dataflow System*. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, pp. 439–455. ISBN: 9781450323888 (cited on pages 155, 158).

Musuvathi, Madanlal, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu (2008). *Finding and Reproducing Heisenbugs in Concurrent Programs*. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, pp. 267–280 (cited on pages 10, 17).

Natarajan, Saranya and David Broman (2018). *Timed C: An Extension to the C Programming Language for Real-Time Systems*. In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 227–239 (cited on page 154).

Nelson, Bruce Jay (1981). *Remote Procedure Call*. PhD thesis. Carnegie Mellon University (cited on page 18).

Nikolov, Hristo, Todor Stefanov, and Ed Deprettere (2008). *Systematic and Automated Multiprocessor System Design, Programming, and Implementation*. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.3, pp. 542–555 (cited on page 137).

Nikolov, Hristo, Mark Thompson, Todor Stefanov, Andy Pimentel, Simom Polstra, R. Bose, Claudiu Zissulescu, and Ed Deprettere (2008). *Daedalus: Toward Composable Multimedia MP-SoC Design*. In: *Proceedings of the 45th Annual Design Automation Conference*. DAC '08. Anaheim, California: Association for Computing Machinery, pp. 574–579. ISBN: 9781605581156 (cited on page 137).

OASIS (Mar. 2019). *MQTT Version 5.0*. OASIS Standard (cited on page 18).

Odendahl, Maximilian, Jeronimo Castrillon, Vitaliy Volevach, Rainer Leupers, and Gerd Ascheid (2013). *Split-cost communication model for improved MPSoC application mapping*. In: *2013 International Symposium on System on Chip (SoC)* (cited on page 140).

OMG (Apr. 2015). *Data Distribution Service*. OMG Specification formal/2015-04-10. Version 1.4 (cited on pages 18, 37).

Orsila, Heikki, Tero Kangas, Erno Salminen, Timo D. Hämäläinen, and Marko Hännikäinen (2007). *Automated Memory-Aware Application Distribution for Multi-Processor System-On-Chips*. In: *Journal of Systems Architecture* 53.11, pp. 795–815. ISSN: 1383-7621 (cited on page 145).

Palencia, Jose C. and M. Gonzalez Harbour (1998). *Schedulability Analysis for Tasks with Static and Dynamic Offsets*. In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pp. 26–37 (cited on page 12).

Panda, Preeti Ranjan (2001). *SystemC: A Modeling Platform Supporting Multiple Design Abstractions*. In: *Proceedings of the 14th International Symposium on Systems Synthesis*. ISSS '01. Montréal, P.Q., Canada: Association for Computing Machinery, pp. 75–80. ISBN: 1581134185 (cited on page 28).

Papazoglou, Mike P. and Willem-Jan van den Heuvel (2007). *Service Oriented Architectures: Approaches, Technologies and Research Issues*. In: *The VLDB Journal* 16.3, pp. 389–415. ISSN: 0949-877X (cited on pages 18, 154).

Parks, Thomas M. (1995). *Bounded Scheduling of Process Networks*. PhD thesis. USA: University of California at Berkeley (cited on pages 19, 21).

Parr, Terence (Jan. 2013). *The Definitive ANTLR 4 Reference*. Ed. by Davidson Pfalzer Susannah. The Pragmatic Programmers. ISBN: 978-1934356999 (cited on page 61).

Pedroni, Volnei A. (2004). *Circuit Design with VHDL*. 3. Cambridge, MA, USA: MIT Press. ISBN: 9780262042642 (cited on page 28).

Pelcat, Maxime, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi (2014). *PREESM: A Dataflow-based Rapid Prototyping Framework for Simplifying Multicore DSP Programming*. In: *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pp. 36–40 (cited on pages 22, 136).

Perrey, Randall and Mark Lycett (2003). *Service-oriented Architecture*. In: *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings*. Pp. 116–119 (cited on pages 18, 154).

Peterson, James L. (1977). *Petri Nets*. In: *ACM Comput. Surv.* 9.3, pp. 223–252. ISSN: 0360-0300 (cited on page 40).

Petri, Carl Adam (1962). *Kommunikation mit Automaten*. PhD thesis. Rheinisch-Westfälisches Institut für instrumentelle Mathematik an der Universität Bonn (cited on page 40).

Pimentel, Andy D., Cagkan. Erbas, and Simon Polstra (2006). *A Systematic Approach To Exploring Embedded System Architectures At Multiple Abstraction Levels*. In: *IEEE Transactions on Computers* 55.2, pp. 99–112 (cited on page 137).

Pivoto, Diego G.S., Luiz F.F. de Almeida, Rodrigo da Rosa Righi, Joel J.P.C. Rodrigues, Alexandre Baratella Lugli, and Antonio M. Alberti (2021). *Cyber-Physical Systems Architectures for Industrial Internet of Things Applications in Industry 4.0: a Literature Review*. In: *Journal of Manufacturing Systems* 58, pp. 176–192. ISSN: 0278-6125 (cited on page 1).

Pop, Antoniu and Albert Cohen (Jan. 2013). *OpenStream: Expressiveness and Data-Flow Compilation of Openmp Streaming Programs*. In: *ACM Transactions on Architecture and Code Optimization* 9.4. ISSN: 1544-3566 (cited on page 22).

Pree, Wolfgang and Josef Templ (2008). *Modeling with the Timing Definition Language (TDL)*. In: *Model-Driven Development of Reliable Automotive Services*. Ed. by Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 133–144. ISBN: 978-3-540-70930-5 (cited on page 33).

Prokopec, Aleksandar (2018). *Pluggable Scheduling for the Reactor Programming Model*. In: *Programming with Actors: State-of-the-Art and Research Perspectives*. Ed. by Alessandro Ricci and Philipp Haller. Cham: Springer International Publishing, pp. 125–154. ISBN: 978-3-030-00302-9 (cited on page 154).

Ptolemaeus, Claudius, ed. (2014). *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org (cited on pages 22, 153).

Puliafito, Antonio, Giuseppe Tricomi, Anastasios Zafeiropoulos, and Symeon Papavassiliou (2021). *Smart Cities of the Future as Cyber Physical Systems: Challenges and Enabling Technologies*. In: *Sensors* 21.10. ISSN: 1424-8220 (cited on page 1).

Quan, Wei and Andy D. Pimentel (2014). *Towards Exploring Vast MPSoC Mapping Design Spaces Using a Bias-Elitist Evolutionary Approach*. In: *2014 17th Euromicro Conference on Digital System Design*, pp. 655–658 (cited on page 145).

Quigley, Morgan, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng (Jan. 2009). *ROS: an Open-source Robot Operating System*. In: vol. 3 (cited on pages 37, 154).

Raynal, Michel and Mukesh Singhal (1996). *Logical Time: Capturing Causality in Distributed Systems*. In: *Computer* 29.2, pp. 49–56 (cited on page 26).

Robison, Arch D. (2013). *Composable Parallel Patterns With Intel Cilk Plus*. In: *Computing in Science & Engineering* 15.02, pp. 66–71. ISSN: 1558-366X (cited on page 13).

Robledo, Julian and Jeronimo Castrillon (2022). *Parameterizable Mobile Workloads for Adaptable Base Station Optimizations*. In: *2022 IEEE 15th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp. 381–386 (cited on page 147).

Roestenburg, Raymond, Rob Williams, and Robertus Bakker (2016). *Akka in Action*. Simon and Schuster. ISBN: 978-1617291012 (cited on pages 15, 94).

Rönngren, Robert and Michael Liljenstam (1999). *On Event Ordering in Parallel Discrete Event Simulation*. In: *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation*. PADS '99. Atlanta, Georgia, USA: IEEE Computer Society, pp. 38–45. ISBN: 0769501559 (cited on page 27).

Rossel, Marcus, Shaokai Lin, Marten Lohstroh, Jeronimo Castrillon, and Andrés Goens (Oct. 2023). *Provable Determinism for Software in Cyber-Physical Systems*. In: *Proceedings of the 15th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. To appear (cited on page 41).

Rovelli, Carlo (2018). *The Order of Time*. Allen Lane. ISBN: 978-0241292525 (cited on page 25).

Roy, Rob and Venkant Bommakanti (Mar. 2017). *Odroid-XU4 User Manual*. Version rev. 20170310. Hard Kernel, Ldt. (cited on page 135).

Saifullah, Abusayeed, Kunal Agrawal, Chenyang Lu, and Christopher Gill (2011). *Multi-core Real-Time Scheduling for Generalized Parallel Task Models*. In: *2011 IEEE 32nd Real-Time Systems Symposium*, pp. 217–226 (cited on page 13).

Samuel, Mingwei (2021). *Hydroflow: A Model and Runtime for Distributed Systems Programming*. MA thesis. EECS Department, University of California, Berkeley (cited on page 155).

Sangiovanni-Vincentelli, Alberto (2002). *Defining Platform-Based Design*. In: *EEDesign of EETimes* 268 (cited on page 2).

Schmidt, Tim, Guantao Liu, and Rainer Dömer (2017). *Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation*. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC '17. Austin, TX, USA: Association for Computing Machinery. ISBN: 9781450349277 (cited on page 29).

Schmuck, Frank B. (Aug. 1988). *The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems*. PhD thesis. Cornell University (cited on page 26).

Schneider, Christian, Miro Spönemann, and Reinhard von Hanxleden (2013). *Just Model! – Putting Automatic Synthesis of Node-Link-Diagrams into Practice*. In: *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pp. 75–82 (cited on page 87).

Schor, Lars, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin-Haeng Kang, and Lothar Thiele (2012). *Scenario-Based Design Flow for Mapping Streaming Applications onto on-Chip Many-Core Systems*. In: *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '12. Tampere, Finland: Association for Computing Machinery, pp. 71–80. ISBN: 9781450314244 (cited on page 137).

Schulze, Christoph Daniel, Miro Spönemann, and Reinhard von Hanxleden (2014). *Drawing Layered Graphs With Port Constraints*. In: *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2, pp. 89–106. ISSN: 1045-926X (cited on page 87).

Schumacher, Christoph, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann (2010). *ParSC: Synchronous Parallel Systemc Simulation on Multi-Core Host Architectures*. In: *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES/ISSS '10. Scottsdale, Arizona, USA: Association for Computing Machinery, pp. 241–246. ISBN: 9781605589053 (cited on page 29).

Schwarzer, Tobias, Andreas Weichslgartner, Michael Glaß, Stefan Wildermann, Peter Brand, and Jürgen Teich (2018). *Symmetry-Eliminating Design Space Exploration for Hybrid Application Mapping on Many-Core Architectures*. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.2, pp. 297–310 (cited on page 144).

Seshia, Sanjit A., Shiyan Hu, Wenchao Li, and Qi Zhu (2017). *Design Automation of Cyber-Physical Systems: Challenges, Advances, and Opportunities*. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.9, pp. 1421–1434 (cited on page 3).

Sha, Lui, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok (2004). *Real Time Scheduling Theory: a Historical Perspective*. In: *Real-Time Systems* 28.2, pp. 101–155 (cited on pages 11, 13).

Sha, Lui, Ragunathan Rajkumar, and John Lehoczky (1990). *Priority Inheritance Protocols: an Approach To Real-Time Synchronization*. In: *IEEE Transactions on Computers* 39.9, pp. 1175–1185 (cited on page 13).

Sheng, Weihua, Stefan Schürmans, Maximilian Odendahl, Mark Bertsch, Vitaliy Volevach, Rainer Leupers, and Gerd Ascheid (2014). *A Compiler Infrastructure for Embedded Heterogeneous MPSoCs*. In: *Parallel Computing* 40.2. Special issue on programming models and applications for multicores and manycores, pp. 51–68. ISSN: 0167-8191 (cited on page 22).

Shibanai, Kazuhiro and Takuo Watanabe (2017). *Actoverse: A Reversible Debugger for Actors*. In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. AGERE 2017. Vancouver, BC, Canada: Association for Computing Machinery, pp. 50–57. ISBN: 9781450355162 (cited on page 17).

[SW] SimPy, Team, *SimPy: Discret Event Simulation for Python*. URL: https://simpy.readthedocs.io/en/latest/, (visited on Dec. 18, 2023) (cited on page 143).

Singh, Amit Kumar, Muhammad Shafique, Akash Kumar, and Jörg Henkel (2013). *Mapping on Multi/Many-Core Systems: Survey of Current and Emerging Trends*. In: *Proceedings of the 50th Annual Design Automation Conference*. DAC '13. Austin, Texas: Association for Computing Machinery. ISBN: 9781450320719 (cited on page 145).

Sirjani, Marjan and Mohammad Mahdi Jaghoori (2011). *Ten Years of Analyzing Actors: Rebeca Experience*. In: *Formal Modeling: Actors, Open Systems, Biological Systems: Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*. Ed. by Gul Agha, Olivier Danvy, and José Meseguer. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 20–56. ISBN: 978-3-642-24933-4 (cited on page 15).

Sirjani, Marjan, Ali Movaghar, Amin Shali, and Frank S. de Boer (2004). *Modeling and Verification of Reactive Systems Using Rebeca*. In: *Fundamenta Informaticae* 63, pp. 385–410 (cited on page 15).

Själander, Magnus, Sally A. McKee, Peter Brauer, David Engdal, and Andràs Vajda (2012). *An LTE Uplink Receiver PHY Benchmark and Subframe-based Power Management*. In: *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pp. 25–34 (cited on page 146).

Sprunt, Brinkley, Lui Sha, and John Lehoczky (June 1989). *Aperiodic Task Scheduling for Hard-Real-Time Systems*. In: *Real-Time Systems* 1.1, pp. 27–60 (cited on page 12).

Stankovic, John A., Insup Lee, Aloysius Mok, and Ragunathan Rajkumar (2005). *Opportunities and Obligations for Physical Computing Systems*. In: *Computer* 38.11, pp. 23–31 (cited on page 3).

Steinberg, Dave, Frank Budinsky, Ed Merks, and Marcelo Paternostro (Dec. 2008). *EMF: Eclipse Modeling Framework*. Second. Addison-Wesley Professional. ISBN: 978-0321331885 (cited on page 85).

Stephens, Robert (1997). *A Survey of Stream Processing*. In: *Acta Informatica* 34.7, pp. 491–541. ISSN: 1432-0525 (cited on page 22).

Stigge, Martin, Pontus Ekberg, Nan Guan, and Wang Yi (2011). *The Digraph Real-Time Task Model*. In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 71–80 (cited on page 12).

Stroustrup, Bjarne (1988). *What Is Object-Oriented Programming?* In: *IEEE Software* 5.3, pp. 10–20 (cited on page 14).

Stuijk, Sander, Marc Geilen, and Twan Basten (2006). *SDF^3: SDF For Free*. In: *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pp. 276–278 (cited on page 141).

Suchert, Felix, Lisza Zeidler, Jeronimo Castrillon, and Sebastian Ertel (July 2023). *ConDRust: Scalable Deterministic Concurrency from Verifiable Rust Programs*. In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs).

Seattle, Washington, USA: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 33:1–33:39. ISBN: 978-3-95977-281-5 (cited on pages 22, 155).
Sutter, Herb (2005). *The Free Lunch Is Over: a Fundamental Turn Toward Concurrency in Software*. In: *Dr. Dobb's journal* 30.3, pp. 202–210 (cited on pages 5, 135).
Tanenbaum, Andrew S. and Robbert van Renesse (1988). *A Critique of the Remote Procedure Call Paradigm*. In: *Proceedings of the European Teleinformatics Conference*. North-Holland, Amsterdam, pp. 775–783 (cited on page 18).
Tang, Yue, Nan Guan, and Wang Yi (2020). *Real-Time Task Models*. In: *Handbook of Real-Time Computing*. Ed. by Yu-Chu Tian and David Charles Levy. Singapore: Springer Singapore, pp. 1–19. ISBN: 978-981-4585-87-3 (cited on page 11).
Tasharofi, Samira, Peter Dinges, and Ralph Johnson (July 2013). *Why Do Scala Developers Mix the Actor Model with other Concurrency Models?* In: vol. 7920, pp. 302–326. ISBN: 978-3-642-39037-1 (cited on pages 15, 59).
Tasharofi, Samira, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha (2012). *TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs*. In: *Formal Techniques for Distributed Systems*. Ed. by Holger Giese and Grigore Rosu. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 219–234. ISBN: 978-3-642-30793-5 (cited on page 17).
Tasharofi, Samira, Michael Pradel, Yu Lin, and Ralph Johnson (2013). *Bita: Coverage-guided, Automatic Testing of Actor Programs*. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 114–124 (cited on page 17).
Tchidjo Moyo, Noel, Eric Nicollet, Frederic Lafaye, and Christophe Moy (2010). *On Schedulability Analysis of Non-cyclic Generalized Multiframe Tasks*. In: *2010 22nd Euromicro Conference on Real-Time Systems*, pp. 271–278 (cited on page 12).
Team SimPy (2023). *SimPy: Discrete Event Simulation for Python* (cited on page 28).
[SW] The Actix Team, *Actix: Actor Framework for Rust*. LIC: Apache-2.0. URL: https://github.com/actix/actix (visited on Nov. 10, 2023) (cited on page 15).
[SW] The KIELER Project, *KIELER Lightweight Diagrams*. LIC: Eclipse Public License 2.0. URL: https://github.com/kieler/KLighD (visited on Nov. 10, 2023) (cited on page 87).
[SW] The KIELER Project, *KLighD for the Web*. LIC: Eclipse Public License 2.0. URL: https://github.com/kieler/klighd-vscode (visited on Nov. 10, 2023) (cited on page 88).
The MISRA Consortium, Roberto Bagnara, Dave Banham, Andrew Banks, Jill Britton, Alex Gilding, Daniel Kästner, Gerlinde Kettl, Michael Rozenau, and Chris Tapp (Sept. 2023). *MISRA C:2023: Guidelines for the Use of the C Language in Critical Systems*, p. 325. ISBN: 1911700081 (cited on page 59).
Theelen, Bart D., Marc C.W. Geilen, Twan Basten, Jeroen P.M. Voeten, Stefan V. Gheorghita, and Sander Stuijk (2006). *A Scenario-aware Data Flow Model for Combined Long-run Average and Worst-case Performance Analysis*. In: *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.* Pp. 185–194 (cited on page 22).
Thiele, Lothar, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang (2007). *Mapping Applications to Tiled Multiprocessor Embedded Systems*. In: *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007)*, pp. 29–40 (cited on page 137).
Thies, William, Michal Karczmarek, and Saman Amarasinghe (2002). *StreamIt: A Language for Streaming Applications*. In: *Compiler Construction*. Ed. by R. Nigel Horspool. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 179–196. ISBN: 978-3-540-45937-8 (cited on page 22).
Thoman, Peter, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos (Apr. 2018). *A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing*. In: *The Journal of Supercomputing* 74.4, pp. 1422–1434 (cited on pages 11, 14).
Thomas, Dave (2018). *Programming Elixir 1.6: Functional|> Concurrent|> Pragmatic|> Fun*. The Pragmatic Bookshelf. ISBN: 9781680502992 (cited on page 15).
Thomas, Donald E. and Philip R. Moorby (2008). *The Verilog Hardware Description Language*. USA: Springer New York. ISBN: 978-0-387-84930-0 (cited on page 28).
Thompson, Mark and Andy D. Pimentel (2013). *Exploiting Domain Knowledge in System-Level Mpsoc Design Space Exploration*. In: *Journal of Systems Architecture* 59.7, pp. 351–360. ISSN: 1383-7621 (cited on page 144).
Thönes, Johannes (2015). *Microservices*. In: *IEEE Software* 32.1, pp. 116–116 (cited on page 18).
Tilkov, Stefan and Steve Vinoski (2010). *Node.js: Using Javascript To Build High-Performance Network Programs*. In: *IEEE Internet Computing* 14.6, pp. 80–83 (cited on page 87).
Tindell, Ken W., Alan Burns, and Andy J. Wellings (Mar. 1994). *An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks*. In: *Real-Time Systems* 6.2, pp. 133–151 (cited on page 12).

Tomlinson, Christine, Won Kim, Mark Scheevel, Vineet Singh, Becky Will, and Gul. Agha (1988). *Rosette: an Object-Oriented Concurrent Systems Architecture*. In: *SIGPLAN Not.* 24.4, pp. 91–93. ISSN: 0362-1340 (cited on page 15).

Torres Lopez, Carmen, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers (2019). *Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper)*. In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 27:1–27:30. ISBN: 978-3-95977-111-5 (cited on page 17).

Torres Lopez, Carmen, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck (2018). *Programming with Actors: State-of-the-Art and Research Perspectives*. In: ed. by Alessandro Ricci and Philipp Haller. Cham: Springer International Publishing. Chap. A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs, pp. 155–185 (cited on page 15).

Trencher, Gregory (2019). *Towards the Smart City 2.0: Empirical Evidence of Using Smartness As a Tool for Tackling Social Challenges*. In: *Technological Forecasting and Social Change* 142, pp. 117–128. ISSN: 0040-1625 (cited on page 1).

Tripakis, Stavros, Christos Stergiou, Chris Shaver, and Edward A. Lee (2012). *A Modular Formal Semantics for Ptolemy*. In: *Mathematical Structures in Computer Science. Accepted for publication* (cited on page 153).

Turing, Aalan M. (1937). *On Computable Numbers, With an Application To the Entscheidungsproblem*. In: *Proceedings of the London Mathematical Society* s2-42.1, pp. 230–265 (cited on page 2).

Van Cutsem, Tom, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter (2014). *AmbientTalk: Programming Responsive Mobile Peer-To-Peer Applications With Actors*. In: *Computer Languages, Systems & Structures* 40.3, pp. 112–136 (cited on page 16).

Veitch, Edward W. (1952). *A Chart Method for Simplifying Truth Functions*. In: *Proceedings of the 1952 ACM National Meeting (Pittsburgh)*. ACM '52. Pittsburgh, Pennsylvania: Association for Computing Machinery, pp. 127–133. ISBN: 9781450373623 (cited on page 6).

Venkataramani, Vanchinathan, Bruno Bodin, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Peh (2020). *Time-Predictable Software-Defined Architecture with SDF-based Compiler Flow for 5G Baseband Processing*. In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1553–1557 (cited on page 146).

Virding, Robert, Claes Wikström, Mike Williams, and Joe Armstrong (1996). *Concurrent Programming in Erlang (2nd Ed.)* GBR: Prentice Hall International (UK) Ltd. ISBN: 013508301X (cited on page 15).

Waxman, Ronald, Jean-Michel Bergé, Oz Levia, and Jacques Rouillard (1996). *High-Level System Modeling*. Springer. ISBN: 9780792396604 (cited on page 4).

Wellings, Andrew (2004). *Concurrent and Real-time Programming in Java*. John Wiley & Sons, Inc. ISBN: 978-0470844373 (cited on page 154).

Wildermann, Stefan, Andreas Weichslgartner, and Jürgen Teich (2015). *Design Methodology and Run-Time Management for Predictable Many-Core Systems*. In: *2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pp. 103–110 (cited on page 145).

Wilhelm, Reinhard, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström (2008). *The Worst-Case Execution-Time Problem-Overview of Methods and Survey of Tools*. In: *ACM Transactions on Embedde Computing Systems* 7.3. ISSN: 1539-9087 (cited on page 57).

Wittig, Robert, Andrés Goens, Christian Menard, Emil Matus, Gerhard P. Fettweis, and Jeronimo Castrillon (Oct. 2020). *Modem Design in the Era of 5G and Beyond: The Need for a Formal Approach*. In: *Proceedings of the 27th International Conference on Telecommunications (ICT)*. Virtual. Bali, Indonesia, pp. 1–5 (cited on pages xi, 146).

[SW] Yadan, Omry, *Hydra—A Framework for Elegantly Configuring Complex Applications* 2019. URL: https://github.com/facebookresearch/hydra (visited on Dec. 18, 2023) (cited on page 145).

Yang, Jixiang and Qingbi He (Apr. 2018). *Scheduling Parallel Computations By Work Stealing: a Survey*. In: *International Journal of Parallel Programming* 46.2, pp. 173–197. ISSN: 0885-7458 (cited on page 155).

Yu, Xinghuo and Yusheng Xue (2016). *Smart Grids: a Cyber-Physical Systems Perspective*. In: *Proceedings of the IEEE* 104.5, pp. 1058–1070 (cited on page 1).

Yuan, Simon, Li Hsien Yoong, and Partha S. Roop (2011). *Compiling Esterel for Multi-core Execution*. In: *2011 14th Euromicro Conference on Digital System Design*, pp. 727–735 (cited on page 31).

Zhao, Yang, Jie Liu, and Edward A. Lee (2007). *A Programming Model for Time-Synchronized Distributed Real-Time Systems*. In: *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pp. 259–268 (cited on pages 6, 52, 82).

Zurstraßen, Niko, José Cubero-Cascante, Jan Moritz Joseph, Li Yichao, Xie Xinghua, and Rainer Leupers (2023). *par-gem5: Parallelizing gem5's Atomic Mode*. In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (cited on page 29).