

UC Berkeley

UC Berkeley Electronic Theses and Dissertations

Title

Verifiable Cyber-Physical Systems: Building Engineering Models for Deterministic and Predictable Concurrent Execution

Permalink

<https://escholarship.org/uc/item/3b3587ft>

ISBN

9798189278860

Author

Lin, Shaokai

Publication Date

2026-05-01

Peer reviewed|Thesis/dissertation

Verifiable Cyber-Physical Systems: Building Engineering Models for Deterministic and
Predictable Concurrent Execution

by

Shaokai Lin

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Edward A. Lee, Co-Chair

Professor Sanjit A. Seshia, Co-Chair

Professor Alberto Sangiovanni-Vincentelli

Professor Stephen A. Edwards

Spring 2026

Verifiable Cyber-Physical Systems: Building Engineering Models for Deterministic and
Predictable Concurrent Execution

Copyright 2026
by
Shaokai Lin

Abstract

Verifiable Cyber-Physical Systems: Building Engineering Models for Deterministic and Predictable Concurrent Execution

by

Shaokai Lin

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Edward A. Lee, Co-Chair

Professor Sanjit A. Seshia, Co-Chair

Cyber-physical systems (CPS) are increasingly deployed in safety-critical domains such as autonomous vehicles, avionics, and industrial automation, yet today's toolchains rarely connect correctness, timing analysis, and implementation in one coherent workflow. Classical timing analysis does not directly support the notion of *logical time*, the abstraction used by deterministic CPS coordination languages; most formal verification tools do not analyze CPS engineering models directly and remain labor-intensive to apply; and hard real-time guarantees often depend on schedules crafted by hand, which is slow and error-prone. *Can we design a toolchain that closes all three gaps in a unified workflow?*

This thesis develops an affirmative answer through a suite of design-time analysis, formal verification, and scheduling techniques for cyber-physical systems, built on the reactor model of computation and the LINGUA FRANCA coordination language.

The contributions are:

- **A system-level timing and trade-off analysis framework** that extends the CAL theorem (consistency, availability, latency) to the reactor model and supports dynamic execution traces. A central proposition expresses each reaction's physical firing time as a max-plus linear recurrence over a system matrix whose eigenvalue identifies the critical path of a distributed program, enabling principled derivation of decentralized coordination parameters in place of manual hand-tuning.
- **A perspective on engineering and scientific modeling in formal verification.** Formal verification reasons only about formal models, so one must distinguish which parts of a model specify intended behavior (engineering models) from which parts

approximate implementations or environments (scientific models), and interpret verification results accordingly. The two roles should not be conflated without keeping that distinction clear. We demonstrate this perspective through an encoding of LINGUA FRANCA’s deterministic semantics in TIMED REBECA, and argue that TR’s timing features are better suited to engineering modeling than to scientific modeling.

- **The LF VERIFIER, the first fully automated verifier for LINGUA FRANCA programs**, which encodes reactor systems’ behavior as axiomatic models sound for all valid runtime implementations, computes bounded model checking completeness thresholds, and automatically translates C reaction bodies into UCLID5 axioms. On a benchmark suite of 22 programs, it correctly verifies 21 fully automatically.
- **State Space Finite Automata (SSFA), a unified formalism for quasi-static scheduling** of deterministic timed concurrent models, subsuming synchronous dataflow, Boolean-controlled dataflow, scenario-aware dataflow, and logical execution time. Integrated into the LINGUA FRANCA compiler and executed by the PRETVM runtime, quasi-statically scheduled programs reduce average runtime overhead by 9× over dynamic scheduling while providing anomaly-free timing and analyzable worst-case makespan without requiring per-step worst-case execution times. An integration with MOCASIN further enables compile-time design space exploration over time–energy trade-offs on heterogeneous platforms.
- **A modernized implementation of GAMETIME**, a timing analysis tool that combines game-theoretic online learning and systematic testing using satisfiability modulo theories (SMT) solvers. GAMETIME can estimate worst-case execution times of reaction bodies as well as answer other timing analysis questions. Rebuilt on LLVM and KLEE and supporting cycle-accurate emulation on FLEXPRET, it supplies timing estimates for sequential programs consumed by the system-level and quasi-static analyses above, closing the loop between design-time reasoning and the actual execution-time behavior of generated code.

These contributions form a toolchain that connects analyzable CPS models, automated verification at the engineering level, and quasi-static scheduling into deterministic, verifiably correct, high-performing implementations.

To my family.

Contents

Contents	ii
List of Figures	iv
List of Tables	vii
1 Introduction	1
1.1 Motivation	2
1.2 Background	4
1.3 Contributions	8
1.4 Outline	9
2 System-Level Timing Analysis for Engineering Models	11
2.1 Related Work	12
2.2 Background	14
2.3 System-Level Timing Analysis Framework	19
2.4 Case Study: Ensuring System Responsiveness	29
2.5 Case Study: Systems with Feedback Loops	33
2.6 Case Study: Deriving Coordination Policy	38
2.7 Conclusion	42
3 Formal Verification of Engineering Models	44
3.1 Engineering Models and Scientific Models	44
3.2 Scientific and Engineering Models in TIMED REBECA	45
3.3 Running Example	48
3.4 Overview of LINGUA FRANCA	52
3.5 Verifying an Engineering Model using TIMED REBECA	56
3.6 Challenges	58
3.7 Future Work	58
4 Formal Verification of LINGUA FRANCA Programs	59
4.1 Running Example: ADAS	62
4.2 Formal Logic and Properties	65

4.3	Axiomatic Semantics for Verifying Reactors	67
4.4	Bounding the Completeness Threshold	71
4.5	LF VERIFIER: the First Fully Automated Verifier for Reactors	77
4.6	Future Work	82
5	Quasi-Static Scheduling for Deterministic Timed Concurrent Models	85
5.1	Background and Related Work	86
5.2	Running Example	88
5.3	Quasi-Static Scheduling based on SSFA	89
5.4	Applying Methodology to the Reactor Model	98
5.5	Evaluation	105
5.6	Design Space Exploration for Heterogeneous Platforms	109
5.7	Conclusion and Future Work	115
6	Timing Analysis of Sequential Programs using GAMETIME	116
6.1	Background	116
6.2	A New GAMETIME with LLVM Support	120
6.3	Checking Path Infeasibility with KLEE	121
6.4	Applications of GAMETIME in this Thesis	125
6.5	Future Work	126
7	Conclusion	128
7.1	Future Work	128
7.2	Final Remarks	131
	Bibliography	132

List of Figures

1.1	An overview of the framework proposed in this thesis.	2
1.2	A minimal LF program with one reactor, one state variable, one periodic timer, and one reaction.	5
1.3	The same reactor augmented with a physical action, an output port, and a second reaction.	6
1.4	A two-reactor LF program with a zero-delay connection from A.out to B.in . . .	7
2.1	A delayed connection can introduce memory across logical tags when its logical delay does not cover the upstream execution time. The downstream reaction at tag k' must wait both for the logical tag to appear and for the upstream reaction at the earlier tag to complete.	27
2.2	The world's smallest stock exchange: a Client reactor receives asynchronous orders via a physical action and forwards them to an Exchange reactor.	30
2.3	Lag measured under the original design, which grows without bound.	32
2.4	Revised stock exchange design with a minimum spacing derived from the max-plus eigenvalue, imposing a rate limit that prevents the exchange from falling behind.	33
2.5	Lag measured under the revised design, which stays bounded.	34
2.6	The Impossible Consistency example: a distributed system of two reactors receiving asynchronous updates while maintaining strong consistency.	34
2.7	Simplified diagram of the Impossible Consistency example, with startup-triggered reactions removed.	35
2.8	A sensor polling system: the Controller sends polling commands to the Sensor via a zero-delay connection and receives responses through a connection with a 1 s logical delay.	39
2.9	Execution trace predicted by Proposition 2.1 for the sensor polling system. When the sensor occasionally exceeds its normal execution time, the controller's maxwait of 121 ms ensures it waits long enough for the delayed response, preserving determinism.	42
3.1	TIMED REBECA model for a ticket service, from [157].	46
3.2	Two accidental deployments of emergency escape slides (image from <i>The Telegraph</i> , Sept. 9, 2015).	48

3.3	Aircraft door system schematic using the Actor model.	49
3.4	TIMED REBECA model for the aircraft door system using priorities.	50
3.5	Using after and delay to avoid accidentally deploying emergency slides, model network latencies, and model execution time.	51
3.6	Aircraft door system schematic and LINGUA FRANCA specification.	53
3.7	TIMED REBECA model encoding the semantics of the LINGUA FRANCA model.	56
4.1	LF code and diagram of a simplified model of an ADAS system.	63
4.2	Verification workflow of LF programs and the architecture of the LF VERIFIER. Blue represents software modules implemented in this work. The other matching colors indicate correspondence between the input program and the generated model.	66
4.3	Logical time and path definitions.	67
4.4	An abstract view of the state space diagram for the running example in Fig. 4.1. In this abstract view, except for S0, which is shown in full, each node consists of (from left to right) an identifier, the number of reactions invoked at the current tag, the number of events pending, and (at the bottom) the current tag.	76
4.5	Syntax for the Safety MTL fragment supported by the LF VERIFIER.	78
4.6	Distribution of benchmark results.	84
5.1	A satellite attitude control system.	88
5.2	Hyperperiod of the satellite attitude controller in Fig. 5.1.	89
5.3	General quasi-static scheduling workflow based on SSFA. The colors, red and blue, refer to two main stages in the workflow: SSFA generation (Sec. 5.3) and DAG scheduling (Sec. 5.3).	89
5.4	DAG representation after conversion from the hyperperiod in Fig. 5.2, showing release edges in blue, deadline edges in red, and data dependency edges in gray, with the virtual path above the dashed line. The node names are the initials of the component names. The numbers in the nodes are the WCET bounds of each task, and the nodes above the dashed line are the nodes on the virtual path. Sync nodes are identified by <i>s</i> , marking the release and deadline of tasks; dummy nodes are identified by <i>d</i> , marking time intervals between sync nodes. Details of virtual nodes are introduced in Sec. 5.3.	90
5.5	Sample output from a DAG scheduler, a transitive reduction of the DAG in Fig. 5.4 with width $w = 2 + 1$, with +1 for the virtual path. The colors of the nodes show a valid partitioning to 2 available cores.	95
5.6	Scheduling anomaly example.	96
5.7	Syntax of the synchronous subset of LF.	99
5.8	Satellite attitude controller with user input and termination, written in the synchronous subset of LF. The C code inside reactions is omitted for conciseness.	100
5.9	State space exploration identifies phases of the updated satellite controller (Fig. 5.8) with guarded transitions. The full-size periodic phase is shown in Fig. 5.2.	102

5.10	SSFA of the updated satellite controller generated from Fig. 5.9. In the Initialization mode, task nodes without WCETs are labeled “-”. In the Termination mode, dummy nodes of “ ∞ ” indicate that tasks do not have timing constraints on completion.	102
5.11	Timing accuracy results showing runtime overhead, measured on QNX (SDP 8.0) running on a Raspberry Pi 4. The X-axis represents reaction names, with each reaction displaying three distributions: DY, LB, and EGS. The Y-axis indicates lag in nanoseconds.	106
5.12	Measured release times and execution times of the reactions from Fig. 5.8 on FLEXPRET.	108
5.13	The proposed DSE tool flow. Dashed boxes denote external tools we build interfaces for.	110
5.14	LF diagram of the file compression service.	112
5.15	Execution time and total energy measured.	113
6.1	CFG and basis paths for the unrolled modular exponentiation code with a 2-bit exponent (from Kotker et al. [90]). (a) The full DAG with 9 edges. (b) Three basis paths $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$. (c) The fourth path $\mathbf{p}_4$ expressed as $\mathbf{p}_1 + \mathbf{p}_2 - \mathbf{p}_3$	118
6.2	Unrolled <code>modexp</code> annotated with condition variable assignments for path $\mathbf{p}_1$ (<code>exponent = 1</code>). Variables <code>cv_0</code> through <code>cv_5</code> (on-path) are initialized to <code>false</code> and set to <code>true</code> ; <code>cv_6</code> and <code>cv_7</code> (off-path) are initialized to <code>true</code> and set to <code>false</code> . When <code>exponent = 1</code> , all variables end up <code>true</code>	123
6.3	Generated KLEE harness for checking feasibility of path $\mathbf{p}_1$. The six on-path condition variables are initialized to <code>false</code> ; the two off-path variables are initialized to <code>true</code> . The wrapper makes <code>base</code> and <code>exponent</code> symbolic, calls <code>modexp</code> , and asserts all eight variables are <code>true</code>	124
6.4	Actual <code>ktest-tool</code> output from a feasibility check on the <code>modexp</code> benchmark. KLEE found concrete inputs <code>base = 0</code> and <code>exponent = 124</code> that drive execution along the candidate path. Paths are abbreviated with <code>...</code> for readability.	125
7.1	Building a satellite reaction wheel controller [180, 38, 181] using the neurosymbolic framework in Figure 1.1	129

List of Tables

4.1	Benchmark results.	83
5.1	Intermediate Representation for Composing Executable Schedules	104
5.2	Average, maximum, and standard deviation of the benchmark performance (lag) using the dynamic scheduler (DY), the static LOAD BALANCED scheduler (LB), and the static EDGE GENERATION scheduler (EGS).	105
5.3	WCET bounds for the reactions in the satellite controller computed by PLATIN.	108
5.4	A subset of mappings for the periodic phase generated by MOCASIN.	111

Acknowledgments

The work in this thesis was supported in part by the National Science Foundation (NSF), awards #CNS-1836601 (Reconciling Safety with the Internet) and #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems) and the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by Denso, Siemens, and Toyota. This work was also supported in part by DARPA grant FA8750-20-C-0156, DARPA Provably Correct Design of Adaptive Hybrid Neuro-Symbolic Cyber Physical Systems (ANSR) program award number FA8750-23-C-0080, and by Intel.

I was immensely fortunate to have received tremendous guidance and help throughout my Ph.D. journey. First and foremost, I owe my greatest gratitude to my co-advisors, Edward A. Lee and Sanjit A. Seshia. In research, Edward and Sanjit consistently gave me crisp and actionable advice when I needed it most. I feel quite fortunate to be advised by the two authors of the Lee & Seshia book on cyber-physical systems [116], as I am deeply inspired by the way they combine mathematical and formal rigor with practical insight into the forefront of embedded systems research. During the past six years at Berkeley, I have also been fortunate to enjoy considerable academic freedom under Edward and Sanjit’s supervision, which I have always appreciated.

Beyond providing research insights, Edward and Sanjit are also role models from whom I learned in a variety of ways. On my first visit to Edward’s house, I discovered that he was also a visual artist. Over the years of interacting with Edward, I observed how he brought his artistic mindset to research. When we stumbled upon an interesting idea during one meeting, I memorably heard Edward say, “Oh, that’s a cute idea.” Another time, I heard him describing a theorem as “beautiful.” These were inspiring experiences, as I realized that there is a difference between “ideas that merely work” and “beautiful ideas.” Hence, in my research, I aimed to search for the latter, rare as they are. I also benefited from reading Edward’s book *Plato and the Nerd*, where concepts such as *abstraction layers* and *engineering models vs. scientific models* significantly shaped the course of my research. Edward’s depth of thinking, articulateness, and productivity never cease to amaze me. On the other hand, a key lesson I learned from Sanjit is to continue having faith in my work. Throughout my Ph.D. journey, I dealt with numerous rejections (for conference papers, fellowships, etc.), which were not easy to process initially. When the LF VERIFIER paper [122] was rejected the second time, I was quite upset, as I thought my work was subpar even after putting in tremendous effort. But during our meeting, Sanjit asked me not to be discouraged and affirmed to me that “This is good work,” which immediately helped me regain confidence. The paper was later accepted at EMSOFT 2023, and to this day, it is one of the papers I am most proud of. I consider myself incredibly lucky to have been able to work with Edward and Sanjit for the past six years.

I also want to thank Alberto Sangiovanni-Vincentelli and Stephen A. Edwards for serving on my committee. I first got to know Alberto in *EE C249B: Embedded System Design: Modeling, Analysis, and Synthesis* in Spring 2021 and was captivated by the course material, eventually working with Alberto as a Graduate Student Instructor (GSI) for the class. Over

the years, Alberto has generously supported me through various milestones in my Ph.D. career, from my qualifying exam and fellowship applications to faculty applications, which I very much appreciate. Alberto’s research impact on industry has also been a source of inspiration for me. Stephen was my research advisor when I was an undergrad at Columbia. I distinctly remember showing up at the CS research fair, where I met Stephen. I expressed my curiosity about pursuing a Ph.D. degree but was on the fence at the time. Stephen warned me to be cautious and recommended the book *Getting What You Came For* by Robert L. Peters, saying “This is the best seven-dollar investment you’ll ever make.” It turned out to be exactly that. Reading the book prepared me for what was to come, and working with Stephen was an exciting and intellectually challenging experience. I therefore thank Stephen for helping me make up my mind about pursuing a Ph.D. after my undergraduate studies, which is one of the most important life decisions I’ve made so far.

I want to thank all my collaborators for all the projects and papers we have worked on together. This thesis would not have been possible without their help and support. I want to thank Marten Lohstroh, who was completing his Ph.D. when I entered the program in 2020 during the peak of the COVID-19 pandemic. Marten was the first person I met at Berkeley in person and introduced me to the world of LINGUA FRANCA and the reactor model. This thesis is, in a way, a continuation of Marten’s thesis work. Besides Marten, founding LF developers such as Christian Menard and Soroush Bateni built a solid software foundation for the LF compiler and runtime, which allowed my early research projects to get off the ground smoothly. Erling Jellum and Tassilo Tanneberger collaborated with me on a variety of projects, from quasi-static scheduling [181], satellite reaction wheel control [180], and LF as an RTOS [80] to building INTERPRET, a multicore CPU microarchitecture [79]. It was fun traversing many layers of abstraction together through these projects. I want to thank Pei-Wei Chen, Adwait Godbole, Sheng-Jung Yu, and Marcus Rossel for exploring how to formally verify LF programs together; we have made tremendous progress on this front since 2020 [122, 38, 159]. Thanks to Yatin A Manerkar for teaching me axiomatic encoding, which has become the foundation of the LF VERIFIER. Mirco Theile and Binqi Sun taught me DAG scheduling fundamentals; without their help, there would not have been the quasi-static scheduling paper [181]. I also want to give credit to Ehsan Khodadad for adapting these ideas to the Patmos processor [162], and to Jiahong Bi, Guangyu Feng, and Yimo Xu for helping integrate the ideas with the MOCASIN framework [180]. Edward Wang was instrumental in helping me start working with the FLEXPRET processor architecture [197]. Sören Domrös and Alexander Schulz-Rosengarten helped me thoroughly understand how LF diagrams work. Kudos to the folks at Kiel University for building such an excellent diagramming framework. Ravi Akella from Denso provided valuable insights from the industry’s perspective. Harun Teper and Shulu Li helped me better understand ROS2, and together we explored synergies between LF and ROS2.

Outside of research, I had a fun time with friends from Europe who were visiting Edward’s lab: Francesco Paladino (now a postdoc), Vincenzo Barbuto, and Matthias Wagner. I also appreciate the friendship and support of Sanjit’s Learn and Verify lab, including Pei-Wei Chen, Adwait Godbole, Aniruddha Joshi, Karim Elmaaroufi, Beyazit Yalcinkaya, Niklas

Lauffer, Justin Wong, Kai-Chun Chang, Kai Xu, Sahil Bhatia, and Ameesh Shah. Thanks also to Karim Elmaaroufi from the L&V lab for working with me on Open Multi-Agent Runtime (OMAR), an open-source project focusing on multi-agent coordination. In addition, I want to thank the 25 brilliant undergraduate & Master’s students I have mentored at Berkeley. I think I have learned more from them than I have taught. The DOP Center (now also known as the SCALE lab) on the fifth floor of Cory Hall is where I spent most of my time in the last six years. I appreciate all my colleagues, in particular Matteo Guarrera, Hanna Krasowski, Alvin Tan, and Tess Despres, with whom I had great hallway chats.

Over the years, I have also been fortunate to interact with and receive support from great faculty members at Berkeley and other universities, including Jeronimo Castrillon, Hokeun Kim, Martin Schoeberl, Linh Thi Xuan Phan, Jian-Jia Chen, Chadlia Jerad, Moez Ben Haj Hmida, Elizabeth Polgreen, Marjan Sirjani, Grzegorz Bazydło, Inigo Incer, Prabal Dutta, Yatin A. Manerkar, and Reinhard von Hanxleden. I especially enjoyed the two-month summer research visit in 2023 with Jeronimo’s group at TU Dresden, where I wrote a lot of the code for PRETVM [121] and made many friends.

At Berkeley, a student organization close to my heart is the Association of Chinese Entrepreneurs (ACE), where I made many inspiring, lifelong friends in the Bay Area. I also appreciate the International Graduate Student Ministry (IGSM), where I received a lot of love and support. One special memory is of Adam Kor from IGSM picking me up from the airport when I first arrived at Berkeley in 2020. I also thank the Cal Japan Club (CJC) for accepting me as a general member, through which I made many Japanese friends, including Yujiro Ota. I also enjoyed studying Japanese at Berkeley for a year and appreciate the friendships I made through the J1A and J1B classes. Outside of Berkeley, I am fortunate to have many Columbia friends in the Bay Area from my undergraduate years. They have become a crucial support network, and I have enjoyed all the fun times spent together.

Lastly and most importantly, none of this would have happened without the love and support from my family: my parents, Julia Zhou and Chunjie Lin, and my brother Fred Lin. My loving grandparents, Wenfu Lin and Lanmei Lin, and my grandmother from my mom’s side, Yuzhe Ke, also gave me tremendous support throughout this journey. Sadly, my grandfather from my mom’s side, Jinpeng Zhou, passed away in 2023. He had been an inspiring figure for me growing up, and I hold him in my memory. Hence, this thesis is dedicated to them.

Chapter 1

Introduction

“Scientific revolutions are here taken to be those non-cumulative developmental episodes in which an older paradigm is replaced in whole or in part by an incompatible new one.”

—*Thomas S. Kuhn, The Structure of Scientific Revolutions, 1962.*

Humans went to the moon again. After five decades, NASA’s Artemis II mission [144] successfully completed a lunar flyby, sending humans farther than ever before and letting us on Earth see the far side of the Moon in 4K. This is a truly exciting time. Compared to Apollo, the software of Artemis has grown tremendously in complexity; because software controls almost all aspects of the Orion spacecraft (thermal, power, etc.), mechanical fallbacks of the Apollo era are no longer an option [95]. To me, Artemis represents the class of systems that intrigue me most: systems that integrate digital computations (the *cyber* world) with physical processes (the *physical* world)—Cyber-Physical Systems (CPS) [116]. Artemis aside, emerging CPS applications such as SpaceX’s Starship, Waymo’s self-driving cars, and Figure’s humanoid robots together paint an enthralling future for our society.

Building cyber-physical systems well requires reasoning about correctness, timing, and deadlines together. A controller must compute the right value, at the right logical instant, and early enough for the physical world to use it. Today’s toolchain falls short on all three dimensions:

1. Classical timing analysis does not directly support the notion of *logical time*, a timing abstraction for computational models that supports deterministic computation.
2. Most formal verification tools do not directly verify CPS engineering models and often require laborious manual effort.
3. Hard real-time guarantees often require manually crafting real-time schedules, a time-consuming and error-prone process.

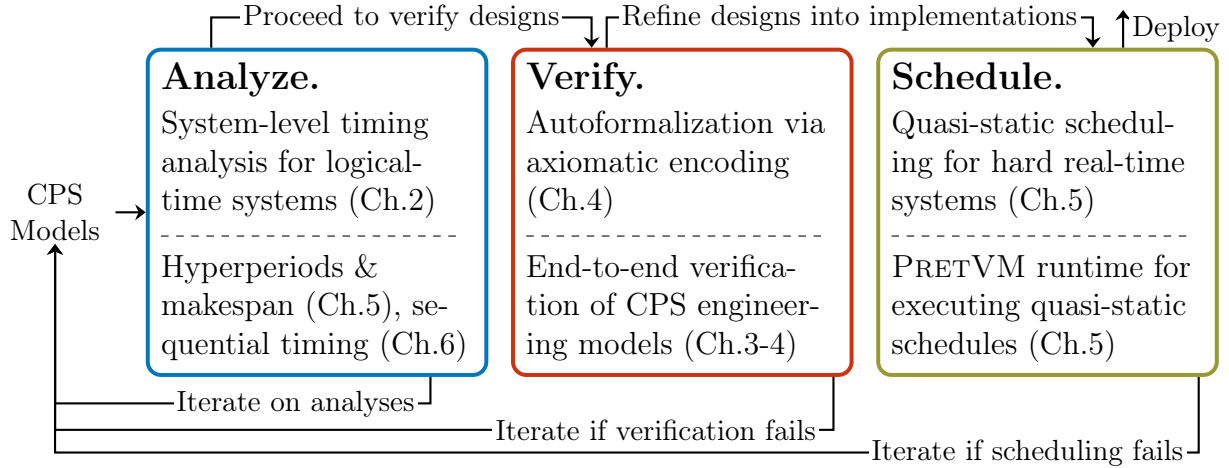


Figure 1.1: An overview of the framework proposed in this thesis.

This thesis explores the question: Can we design a toolchain that closes all three gaps in a unified workflow? The answer developed in this thesis is a framework of abstractions and tools for iterative enhancement across analysis, verification, and scheduling, yielding deterministic, verifiably correct, and high-performing CPS implementations. To pursue this vision, we explore research problems around three main themes, shown in Figure 1.1:

1. system-level timing analysis for logical-time systems,
2. automated verification of CPS engineering models, and
3. a quasi-static scheduling technique with hard real-time guarantees.

We are advancing this vision by building upon the LINGUA FRANCA language as an analyzable engineering abstraction, extending the *CAL theorem* into a framework for system-level timing and trade-off analysis, developing automated verification for reactor-based CPS models, and introducing the *State Space Finite Automata* formalism for quasi-static scheduling.

1.1 Motivation

Timing Analysis for Logical-Time Systems

Modern CPS coordination languages such as LINGUA FRANCA (LF) [128] use *logical time* to express causal structure, simultaneity, timers, and interaction with asynchronous physical events. Two timelines are at play in such models: a logical timeline that specifies when events are intended to occur, and a physical timeline that records when they actually occur during execution. The language relates the two by letting engineers specify how much deviation

from logical time is tolerable before an execution is considered faulty. The resulting model says not only how long computation may take, but also in what order events are allowed to be processed, which reactions are simultaneous, and which outputs are timely enough to be meaningful. Timing analysis for real-time software traditionally begins with tasks, execution-time bounds, and physical deadlines. These ingredients remain necessary, but they are insufficient for modern coordination languages like LF because such analysis reasons about the physical timeline alone. *Can timing analysis be extended to include logical time, so that modern logical-time-based engineering models can be rigorously analyzed at design time?* In this thesis, we focus on LINGUA FRANCA (LF), an open-source¹ coordination language for CPS, and extend LF’s analytical capabilities on system-level timing and on the timing analysis of sequential programs, to ensure the timing correctness of the CPS engineering model *before* code generation.

This logical-time abstraction makes it possible to ask timing questions at the level of the CPS model itself. In distributed CPS, one such question is how coordination delays affect the trade-off between consistency, availability, and latency. The famous CAP theorem [25] states that any networked shared-data system can only obtain two of the three fundamental properties: **c**onsistency, **a**vailability, and **p**artition tolerance. *Can we quantify the exact amount of trade-offs among these properties our engineering model entails?* We pursue this goal by building on the CAL theorem [109, 112, 113] (**c**onsistency, **a**vailability, **l**atency), where latency quantifies the degree of partition tolerance. Our proposed framework establishes a dynamic algebraic relationship between the three properties over an execution trace, enabling practical system-level timing and trade-off analysis in CPS engineering models (Chapter 2). Besides reasoning about trade-offs, ensuring timing correctness is critical as CPS operate in real-time with the physical environment. *Can we compute model-level timing properties such as hyperperiods, anomaly freedom, and worst-case makespan directly from CPS engineering models?* At the same time, model-level timing analysis ultimately depends on how long the target-language procedures inside reactions may take. *Can we estimate worst-case execution times of sequential programs in a way that feeds back into system-level timing analysis?* Toward these aims, we develop analytical tools to reason about crucial properties such as systems’ hyperperiods, conditions for anomaly-free timing, linear-time worst-case makespan analysis, and worst-case execution time estimates for sequential programs (Chapter 5 and Chapter 6). These techniques have built-in tooling support in the LINGUA FRANCA language and can generalize across a wide range of models of computation.

Automated Formal Verification of CPS Engineering Models

For CPS applications that are safety-critical, it is crucial to ensure that high-level specifications correctly align with the engineer’s intent. Formal methods have matured as an effective technique to thoroughly detect errors in system designs. *Can we seamlessly integrate formal methods into CPS engineering by automatically verifying engineering models?*

¹Project website: www.lf-lang.org

A second, equally important challenge is that real-life CPS applications often have reaction bodies written in languages such as C, which typically require manual translation into a formal language for verification. This manual effort is often the bottleneck to formal methods’ usability. *Can programs be automatically formalized to facilitate an automated verification workflow?* To answer both questions, we lead the development of the LF VERIFIER [122] (Chapter 4), which encodes an LF program together with a Metric Temporal Logic (MTL) property as a semantically equivalent *axiomatic model* and discharges the resulting verification obligation to the UCLID5 formal modeling and verification system [171, 152, 184]. The LF VERIFIER further automates the translation of C reaction bodies into UCLID5 axioms, yielding the first end-to-end automated verifier for reactor systems.

Quasi-Static Scheduling with Hard Real-Time Guarantees

After a CPS engineering model is analyzed and verified, the next key challenge is producing an implementation that *preserves* its timing and correctness properties. For hard real-time systems, this step has often depended on hand-crafted schedules: experts translate a model into a predictable execution order, then tune that schedule against platform-specific timing constraints. This practice can produce excellent systems, but it does not scale to rapid iteration or provide a systematic link from model-level guarantees to executable code. *Can schedules be synthesized directly from the model, so that hard real-time implementations are predictable by construction?* Our work addresses this gap through *correct-by-construction* refinement that compiles verified models into implementations with guarantees enforced statically. Specifically, we develop *quasi-static scheduling* techniques that derive real-time schedules directly from CPS engineering models. The *State Space Finite Automata* (SSFA) formalism (Section 5.3) unifies scheduling across a range of deterministic timed concurrent models of computation and ensures analyzable timing without requiring worst-case execution times at every step. Scheduled SSFA programs are lowered to an instruction-level intermediate representation interpreted at runtime by PRETVM (Section 5.4), a lean virtual machine layer that executes quasi-static schedules with analyzable worst-case execution time (WCET). An integration with MOCASIN (Section 5.6) further enables compile-time schedules that explore time–energy trade-offs on heterogeneous platforms. WCET bounds for sequential programs for individual reaction bodies, which the scheduling framework consumes, are produced by a modernized implementation of GAMETIME (Chapter 6), a timing analysis tool based on machine learning from measurements for tests generated by SMT-based analysis, built on top of LLVM and KLEE [32] and supporting cycle-accurate emulation on FLEXPRET [197] as one of its backends.

1.2 Background

Because the analyses in this thesis are built around LINGUA FRANCA (LF), we first introduce the core LF concepts through a small program. LF is a coordination language: it describes

```

1 target C
2
3 reactor A {
4   state s:int = 0
5   // offset = 0, period = 1s
6   timer t(0, 1s)
7   reaction(t) {=
8     /* C code here */
9     self->s++;
10  =}
11 }
12
13 main reactor {
14   a = new A()
15 }

```

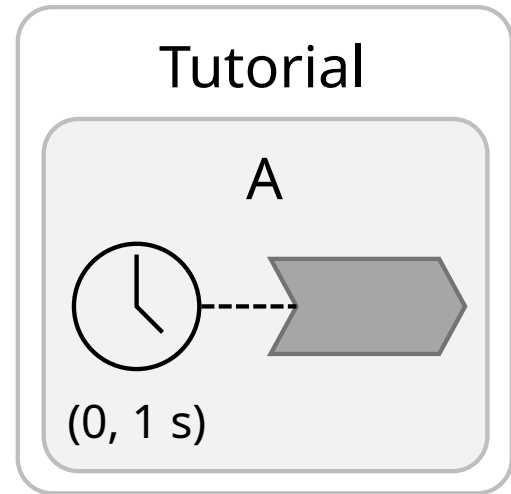


Figure 1.2: A minimal LF program with one reactor, one state variable, one periodic timer, and one reaction.

how stateful components interact in logical time, while allowing the computation inside those components to be written in an ordinary programming language. The program in Figure 1.2 targets C, so the body of its reaction contains C code.

The fundamental component in LF is a *reactor*. A reactor is a stateful container with a well-defined interface and a set of message handlers, called *reactions*. In this example, line 3 declares reactor **A**, and line 4 gives it one state variable, **s**, initialized to zero. The state belongs to the reactor instance and persists across reaction invocations.

The declaration **timer t(0, 1s)** on line 6 creates a periodic logical-time event source. Its first event has offset zero, and subsequent events occur every one second of logical time. The reaction **reaction(t)** on line 7 is triggered by this timer: whenever **t** produces an event, the reaction body executes. Because the program targets C on line 1, the code between **{=** and **=}** is C code; here, line 9 increments the reactor's state variable. More generally, reactions can be triggered by timers, input ports, actions, or other events, and their bodies can use the target language selected by the program.

An LF program also needs a *main reactor*, declared on line 13, which defines the top-level composition to execute. The main reactor in Figure 1.2 creates one instance of reactor **A** on line 14. Although this program has no connections yet, it already illustrates the basic LF structure used throughout the thesis: reactors encapsulate state, triggers determine when reactions run, and target-language code implements the local computation.

We can extend the example with an asynchronous event source and an output port, as shown in Figure 1.3. The declaration on line 12 introduces a *physical action*, which represents an asynchronous event from the physical environment. Unlike the timer, whose events

```

1 target C
2
3 reactor A {
4   state s:int = 0
5   // offset = 0, period = 1s
6   timer t(0, 1s)
7   reaction(t) {=
8     /* C code here */
9     self->s++;
10  =}
11  // asynchronous event
12  physical action a
13  output out:int
14  reaction(a) -> out {=
15    /* C code here */
16    lf_set(out, self->s);
17  =}
18 }
19
20 main reactor {
21   a = new A()
22 }

```

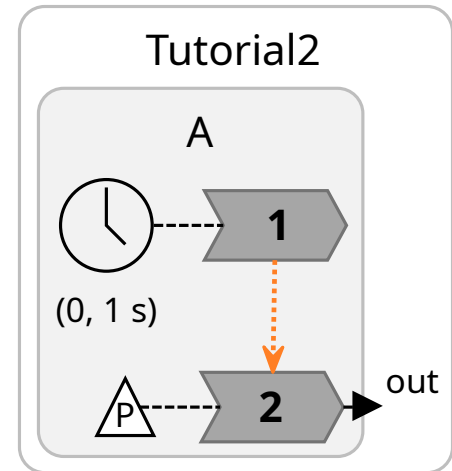


Figure 1.3: The same reactor augmented with a physical action, an output port, and a second reaction.

are generated periodically in logical time, a physical action can be triggered by something external to the LF program, such as an interrupt, a sensor callback, or another runtime event source. The declaration on line 13 introduces an *output port*, through which the reactor can publish values to other reactors. The second reaction, declared on line 14, is triggered by the physical action **a** and has permission to write to the output port **out**, indicated by **-> out**. Inside the C reaction body, line 16 uses the LF API **lf_set** to set the value of the output port to the current state **s**.

The two reactions in Figure 1.3 also illustrate reaction ordering within a reactor. The timer-triggered reaction on line 7 updates **s**, and the physical-action-triggered reaction on line 14 reads **s** and writes it to **out**. Because the second reaction depends on the state possibly updated by the first, LF orders the two reactions deterministically when both are triggered at the same logical tag: reaction 1 executes before reaction 2.

Finally, we can add a second reactor and connect the two reactors, as shown in Figure 1.4. Line 19 declares another reactor, **B**. Its input port, declared on line 20, receives integer values from other reactors. The reaction on line 21 is triggered when a message is present on that input port. Inside the C reaction body, line 23 reads the input value through **in->value** and

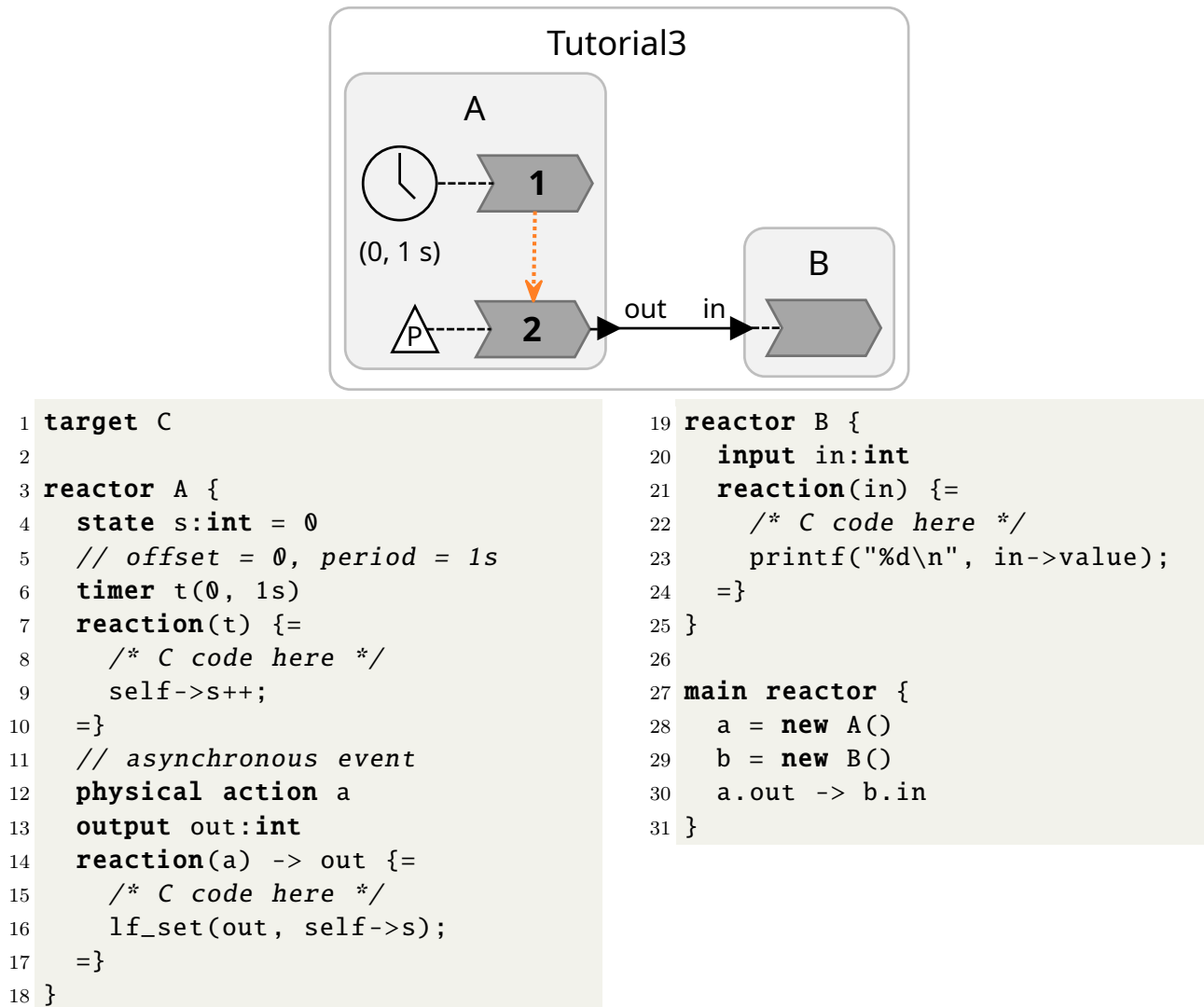


Figure 1.4: A two-reactor LF program with a zero-delay connection from **A.out** to **B.in**.

prints it. The main reactor now creates two reactor instances: one instance of **A** on line 28 and one instance of **B** on line 29. The connection statement on line 30 connects the output port **out** of **A** to the input port **in** of **B**. Because the connection has no delay annotation, it is a *zero-delay* connection: when **A** produces a message on **out**, reactor **B** receives that message at the same logical tag on **in**.

The above example provides a basic introduction to the LINGUA FRANCA language and the reactor model. In subsequent chapters, more details will be introduced where they become relevant.

1.3 Contributions

This thesis makes contributions across design-time analysis, formal verification, and correct-by-construction refinement of cyber-physical systems. The contributions are as follows.

The first contribution is a *system-level timing analysis framework* tailored to the reactor model of computation and the LINGUA FRANCA coordination language (Chapter 2). The technical core is a central proposition that expresses the physical firing time of every reaction as a max-plus linear recurrence over a structured system matrix whose entries encode the causal, dataflow, and timing constraints of a federated LINGUA FRANCA program. A factorization of this matrix exposes the trade-off between consistency, availability, and latency at every reaction, revealing which incoming dependency is the binding constraint on its firing time and quantifying the consistency one would have to relinquish to recover availability. The eigenvalue of the system matrix characterizes the *critical path* of a distributed reactor program, identifying the dominant feedback loop whose cycle weight governs steady-state throughput. We demonstrate through case studies that the framework enables systematic derivation of *maxwait* parameters (safe-to-advance and safe-to-assume-absent offsets) for decentralized coordination directly from trace data, replacing conservative hand-tuning with principled, measurement-driven values.

The second contribution is a philosophy and methodology for formal verification of CPS models that keeps *engineering modeling* and *scientific modeling* distinct (Chapter 3). Engineering models are specifications of correct behavior; scientific models approximate physical implementations and environments. Because formal verification reasons about formal models, it is important to identify which aspects of a model play which role and not to conflate the two roles without keeping that distinction clear. We observe that models used in practice sometimes mix these roles and that such conflation can compromise the trustworthiness of verification results. We demonstrate this perspective concretely by showing how TIMED REBECA, extended with global priorities for message handlers, can faithfully encode the deterministic reaction ordering of LINGUA FRANCA programs, thereby enabling model checking of safety properties that hold under the LINGUA FRANCA semantics regardless of physical execution times and network delays.

The third contribution is the LF VERIFIER, the first fully automated verifier for reactor systems (Chapter 4). The LF VERIFIER defines a verification problem of checking Safety Metric Temporal Logic (Safety MTL) formulas for LINGUA FRANCA programs and encodes the execution of an LF program as an *axiomatic model* rather than an operational one. Because axiomatic models specify behavior based on the LINGUA FRANCA semantics directly, they are sound with respect to *all* valid runtime implementations that correctly implement those semantics. The chapter further presents an algorithm that computes an upper bound on the bounded model checking *completeness threshold* given an LF program and a Safety MTL property, and develops an automatic translation of C reaction bodies into UCLID5 axioms. The resulting tool is embedded in the LINGUA FRANCA compiler: the user need only add **@property** annotations to an LF program, and the verifier automatically generates the UCLID5 model, invokes the Z3 solver, and reports the result. An evaluation on a bench-

mark suite of 22 programs sampled from real-life applications shows that the LF VERIFIER correctly checks 21 out of 22 programs fully automatically.

The fourth contribution is the *State Space Finite Automata* (SSFA) formalism and the first unified approach to quasi-static scheduling for deterministic timed concurrent models (Chapter 5). SSFA serves as an intermediate representation that bridges the gap between high-level models of computation and low-level executable schedules. We identify a class of models called *deterministic, timed, concurrent models* (DTCMs) to which the SSFA-based methodology applies, unifying scheduling techniques previously developed in isolation for synchronous dataflow, Boolean-controlled dataflow, scenario-aware dataflow, and logical execution time. The framework provides anomaly-free timing guarantees without requiring worst-case execution times and enables analyzable worst-case makespan at compile time. We integrate the approach into the LINGUA FRANCA compiler, where quasi-statically scheduled programs reduce average runtime overhead by $9\times$ compared to dynamically scheduled programs. Additionally, we extend the framework with design space exploration for heterogeneous platforms through an integration with MOCASIN, enabling compile-time schedules that encode user-specified trade-offs between execution time and energy consumption.

The fifth contribution is a modernized implementation of GAMETIME [168, 169, 170], a measurement-based timing analysis tool that estimates worst-case execution times (WCETs) of sequential programs using basis-path methods (Chapter 6). The original GAMETIME relied on the now-discontinued CIL and Microsoft Phoenix compiler front-ends, limiting it to C programs. We re-implement GAMETIME on top of LLVM and Clang, replacing the custom symbolic execution engine with KLEE [32] for path feasibility checking. The new implementation introduces a condition-variable instrumentation scheme that constrains KLEE to explore a single target path, enabling precise feasibility verdicts and concrete test-input generation directly from LLVM IR bytecode. The resulting tool can, in principle, analyze any program that compiles to LLVM IR and supports multiple measurement backends, including cycle-accurate emulation on FLEXPRET. We demonstrate how GAMETIME supplies WCET estimates for sequential programs consumed by the system-level timing analysis of Chapter 2 and the quasi-static scheduling framework of Chapter 5, closing the loop between system-level timing analysis and the actual execution-time behavior of generated code.

1.4 Outline

The remainder of this thesis is organized as follows.

Chapter 2 presents a system-level timing analysis framework that refines the original CAL theorem to the reactor model of computation. We formalize the trade-off between consistency, availability, and latency at the granularity of individual reaction firings, state a central proposition that expresses firing times as a max-plus linear recurrence, and show how the resulting precision enables the derivation of execution policies for distributed coordination in LINGUA FRANCA. This chapter is joint work with Edward A. Lee.

Chapter 3 develops the philosophical and technical foundations for verifying CPS engineering models. We distinguish engineering models from scientific models, argue that the two styles should not be conflated without a principled separation, and explain how verification conclusions depend on which role each part of a formal model plays. We demonstrate this perspective concretely through an encoding of LINGUA FRANCA’s timing semantics in TIMED REBECA. This chapter draws from and expands on previously published work titled “Verify Engineering Models, not Scientific Models” [120], co-authored with Edward A. Lee.

Chapter 4 presents the LF VERIFIER, the first fully automated verifier for LINGUA FRANCA programs. The chapter develops axiomatic models that encode LF program behavior at the semantics level, defines a completeness threshold algorithm for bounded model checking, and describes the automatic translation of C reaction bodies into formal axioms. This chapter draws from and expands on previously published work titled “Towards Building Verifiable CPS using LINGUA FRANCA” [122] co-authored with Yatin A Manerkar, Marten Lohstroh, Elizabeth Polgreen, Sheng-Jung Yu, Chadlia Jerad, Edward A. Lee, and Sanjit A. Seshia.

Chapter 5 introduces the State Space Finite Automata formalism and the unified quasi-static scheduling framework. We define the class of deterministic timed concurrent models to which the approach applies, establish key properties including anomaly-free timing and analyzable worst-case makespan, evaluate the approach through benchmarks and a satellite attitude controller case study, and extend it with design space exploration for heterogeneous platforms via MOCASIN. This chapter draws from and expands on previously published work titled “Quasi-Static Scheduling for Deterministic Timed Concurrent Models on Multi-Core Hardware,” co-authored with Erling Jellum, Mirco Theile, Tassilo Tanneberger, Binqi Sun, Chadlia Jerad, Yimo Xu, Guangyu Feng, Magnus Mæhlum, Jian-Jia Chen, Martin Schoeberl, Linh Thi Xuan Phan, Jeronimo Castrillon, Sanjit A. Seshia, and Edward A. Lee, as well as “Navigating Time and Energy Trade-offs in Reactive Heterogeneous Systems,” co-authored with Tassilo Tanneberger, Jiahong Bi, Guangyu Feng, Yimo Xu, Julian Robledo, Robert Khasanov, and Jeronimo Castrillon.

Chapter 6 presents GAMETIME, a measurement-based timing analysis tool that uses basis-path methods to estimate worst-case execution times without requiring detailed hardware models. We describe the modernized LLVM-based implementation and show how GAMETIME supplies timing estimates for sequential programs consumed by the analyses in earlier chapters. This chapter is joint work with Haron Benfaida, Abdalla Mahgoub Eltayeb, Colin Cai, Andrew Zhang, Edward A. Lee, and Sanjit A. Seshia.

Finally, the last chapter concludes the thesis with a discussion of future directions, including automating the proposed framework in Figure 1.1 with tool-using AI, extending trade-off and timing analyses, and generalizing to new domains such as the Robot Operating System, time-predictable networks, and deterministic machine learning.

Chapter 2

System-Level Timing Analysis for Engineering Models

“In a distributed system, it is sometimes impossible to say that one of two events occurred first.”

—Leslie Lamport, *“Time, Clocks, and the Ordering of Events in a Distributed System,”* 1978.

Distributed cyber-physical systems are notoriously difficult to design correctly. A sensor reading on one node must be processed and acted upon at another node within a bounded physical time, yet the designer faces a web of interacting delays: execution times that vary with input data and platform load, network latencies that fluctuate with traffic and routing, and clock synchronization errors that shift the very notion of “now” across nodes. Without a principled framework, the only recourse is conservative over-provisioning or, worse, trial-and-error tuning that breaks silently when operating conditions change.

The *CAL theorem* [112, 109], where C stands for consistency, A for availability, and L for latency, provides such a framework. It replaces the qualitative, all-or-nothing perspective of Brewer’s CAP theorem [26, 62] (consistency, availability, partition tolerance) with a quantitative relationship: for every link in a distributed system, the unavailability (physical-time delay before a node processes an event) is bounded by the apparent network latency minus the inconsistency (logical-time skew) that the designer is willing to tolerate. By expressing this relationship as a linear equation in the max-plus algebra, the CAL theorem turns a long-standing piece of distributed-systems folklore into a design-time calculus for distributed real-time systems.

This chapter refines the original CAL theorem into a version tailored for the reactor model. Whereas the original CAL theorem algebraically relates *worst-case* bounds on consistency, availability, and latency that must hold across an entire execution, the framework developed in this chapter operates at the granularity of individual reactions and models their *exact* physical firing times along a specific execution trace. Each reaction’s firing time

is expressed as a max-plus linear function of input tags, timestamps, measured execution times, and measured network latencies, which collapses the pessimism inherent in worst-case bounds and lifts the CAL relation from a system-wide inequality to a per-reaction equality.

Challenges. The original CAL theorem reasons about fundamental bounds that hold across an entire execution. Because it takes worst-case values for execution times, network latencies, and clock errors, the resulting bounds are necessarily pessimistic: they must account for the worst combination of delays that could ever arise, even if that combination rarely or never occurs in practice. This pessimism has two concrete consequences. First, it prevents the theorem from making use of the detailed, per-event execution traces that are routinely collected from running systems; since the traces contain exact timing data, not worst-case envelopes, the original theorem has little to say about them. Second, it forecloses analyses that exploit structure in the timing data, such as correlations between successive execution times or periodic patterns in network latency, because every quantity is collapsed to a single supremum.

Our approach. Our framework addresses both limitations. By accepting exact, per-event timing data as input, it can analyze recorded traces to explain observed behavior (e.g., why a particular reaction fired late) and can incorporate richer timing models (e.g., execution times drawn from a distribution or measured profile) to make precise predictions about future behavior. The resulting max-plus linear recurrence admits closed-form analysis: the Kleene star of the system’s delay matrix yields the processing offsets needed for decentralized coordination (Section 2.6), and the max-plus eigenvalue of the system matrix identifies the critical path that limits throughput (Section 2.4). These quantities directly determine the `maxwait` parameters used by LINGUA FRANCA’s decentralized coordinator, turning algebraic analysis into executable configuration.

2.1 Related Work

The framework developed in this chapter sits at the intersection of several long-standing research threads: the quantification of fundamental trade-offs in distributed systems, the separation of logical and physical time in cyber-physical system design, and the use of max-plus algebra to analyze timed discrete-event systems. We review each in turn and position our framework with respect to them.

The CAP theorem and its quantifications. Brewer’s CAP conjecture [26], formalized by Gilbert and Lynch [62], states that a networked shared-data system can provide at most two of consistency, availability, and partition tolerance. In a 2012 retrospective, Brewer [25] clarified that partition tolerance is not an independent design choice: partitions are unavoidable, so the question is how to trade consistency for availability when they occur. Abadi [1] observed that CAP says nothing about the common case when the network is

not partitioned, and proposed PACELC, which additionally relates consistency to latency during normal operation. The original CAL theorem [112, 109] unifies Brewer’s and Abadi’s positions by replacing partition tolerance with a numerical measure of apparent latency and deriving an algebraic relation, linear in a max-plus algebra, that relates inconsistency, unavailability, and latency. CAP is then recovered as the limiting case in which latency grows unbounded.

Earlier attempts to quantify CAP focused on probabilistic or combinatorial notions of consistency and availability. Yu and Vahdat [192] quantify availability as the fraction of satisfied accesses and inconsistency as the fraction of out-of-order writes, and show that computing availability as a function of consistency is NP-hard. Rahman et al. [155] derive a CAL-style relation for the special case of two nodes communicating with perfectly synchronized clocks. The CAL theorem generalizes both lines of work to arbitrary heterogeneous network topologies with imperfect clocks and gives a strikingly simple linear relation in a max-plus algebra. The framework developed in this chapter is a further refinement: whereas the original CAL theorem reasons about worst-case bounds that must hold across an entire execution, it operates at the granularity of individual reactions and reasons about the *exact* physical firing times along a specific trace. This exactness allows the framework to use measured execution times and network latencies, rather than conservative upper bounds, and thereby to derive tight execution policies (such as processing offsets used by LINGUA FRANCA’s decentralized coordinator) that would otherwise be hidden behind worst-case pessimism.

Logical time, physical time, and deterministic coordination. Both the original CAL theorem and the framework developed in this chapter rely on a careful separation between logical (tagged) time and physical (clock) time. This separation has a long history in embedded and cyber-physical system design. Lamport’s logical clocks [98] established the “happened-before” relation and showed how timestamps can be used to define a total order of events without relying on synchronized physical clocks. Schwartz and Mattern [166] build on this foundation to define causal consistency and consistent global snapshots, which serve as the event model underlying the CAL theorem. The synchronous languages [17, 65] crystallize this idea in a programming model where computation is logically instantaneous and reacts to a stream of input events at well-defined logical instants. The logical execution time (LET) paradigm [68, 85] and its system-level extension for distributed real-time software [59] decouple the logical timing of task inputs and outputs from the physical execution time, yielding time-deterministic behavior across platforms. LINGUA FRANCA [129, 130, 123] combines these ideas with a reactor-based coordination model [126] that assigns a tag to every event and guarantees that reactions fire in tag order. Our framework inherits this event model directly and expresses reaction firing times as max-plus linear functions of input tags, timestamps, and physical delays. Discussions of nondeterminism in distributed CPS coordination [15] and of the role of logical time in deterministic concurrency [125] further motivate the distinction we exploit.

Max-plus algebra for timed systems. Max-plus algebra has been used for decades to analyze synchronization-dominated discrete-event systems. Seminal work by Cohen et al. [39] and the textbook treatments of Baccelli et al. [12], Cuninghame-Green [42], and Heidergott et al. [67] show that timed event graphs admit a linear representation in the $(\max, +)$ semiring, enabling closed-form expressions for throughput, cycle time, and steady-state behavior via spectral analysis of max-plus matrices. Surveys such as De Schutter et al. [165] and Komenda et al. [89] summarize how these techniques are applied to manufacturing systems, transportation networks, and digital hardware pipelines. More recently, max-plus algebra has been used to derive safe upper bounds on response times in distributed control systems [2] and to reason about weighted timed event graphs [87]. The original CAL theorem [112, 109] frames the CAP trade-off as a max-plus linear relation; our framework takes this algebraic viewpoint further by building a structured system matrix whose factorization reveals Proposition 2.1 at every reaction and whose eigenvalue characterizes critical paths of the distributed reactor program.

Analysis and verification of reactor-based CPS. Our work is also complementary to a growing body of analysis tools for reactor-based cyber-physical systems. Federated LINGUA FRANCA [16] provides predictable coordination for safety-critical distributed embedded systems, and recent work on verifiable CPS in LINGUA FRANCA [122, 175] applies model checking and formal semantics [136, 159] to reactor programs. Debugging and analysis environments [44] offer traces that our framework can ingest directly, since its trace-level analysis consumes exact firing times rather than worst-case bounds. To the best of our knowledge, none of these tools provide a quantitative algebraic account of the consistency–availability–latency trade-off at the granularity of individual reactions, nor do they derive decentralized coordination parameters (such as maxwaits) from such an account. The framework developed in this chapter fills that gap.

2.2 Background

The CAP Theorem

Brewer’s CAP theorem famously states that “any networked shared-data system can have at most two of the three desirable properties: 1. *consistency* (C) equivalent to having a single up-to-date copy of the data; 2. *high availability* (A) of that data (for updates); and (3) tolerance to network *partitions* (P)” [25]. First articulated as a conjecture by Eric Brewer in his 2000 PODC keynote [26], the principle crystallized a tension that had long been felt by practitioners building wide-area and internet-scale services but had rarely been stated with such clarity. Two years later, Gilbert and Lynch [62] transformed the conjecture into a formal impossibility result, proving within an asynchronous network model that no distributed data store can simultaneously guarantee linearizable consistency, total availability for every non-failing node, and continued operation in the face of arbitrary message loss between nodes.

Their proof is short and elegant: assume a system that claims all three properties; partition two replicas so no messages flow between them; issue a write to one side and an immediately following read to the other; availability forces both operations to terminate, but consistency requires the read to observe the write, which is impossible without communication. CAP thus joined FLP [54] as one of the canonical impossibility results delimiting what distributed systems can achieve.

In the formal model, each of the three properties has a precise meaning that is easy to misread in casual prose. *Consistency* in CAP refers specifically to *atomic* or *linearizable* consistency: there exists a total order on all operations such that each operation appears to take effect instantaneously at some point between its invocation and its response, and every read returns the value of the most recently committed write in that order. This is strictly stronger than the “C” of ACID transactions, which concerns preservation of application-level invariants. *Availability* means that every request received by a non-failing node must result in a (non-error) response; crucially, the definition admits no bound on latency, so a system that eventually answers is still considered available in the formal sense. *Partition tolerance* means that the system continues to function, that is, continues to satisfy its remaining guarantees, even when the network drops or arbitrarily delays an unbounded number of messages between nodes. Partitions here are not catastrophic hardware failures but mundane phenomena: switch reboots, congested links, garbage-collection pauses, and asymmetric routing all manifest at the application layer as partitions.

The popular gloss of CAP as “pick any two of the three” is, as Brewer himself later emphasized, an oversimplification that has caused significant confusion [25]. In any real distributed system that spans more than a single fault domain, partitions will occur; the designer therefore does not have the luxury of “choosing” to forgo P. The genuine design choice is conditional: *when a partition occurs*, does the system sacrifice availability to preserve consistency (a CP choice), or sacrifice consistency to preserve availability (an AP choice)? In the absence of partitions, systems can and do provide both C and A, and much of the engineering art lies in detecting partitions quickly, reasoning about the *partition mode* in which reduced guarantees apply, and designing principled *recovery* procedures that restore invariants once connectivity returns. Brewer’s retrospective twelve years later [25] reframes CAP around exactly this life cycle of partition detection, degraded operation, and post-partition reconciliation, rather than as a static trichotomy.

The CAL Theorem

Brewer’s CAP theorem [26, 62] is a foundational result in distributed systems: in the presence of network *partitions* (P), a distributed system must sacrifice either *consistency* (C), meaning agreement on the values of shared state, or *availability* (A), meaning the ability to respond to requests that read or write that shared state. While immensely influential, the CAP theorem is a qualitative, binary result: partitioning is treated as all-or-nothing, and consistency and availability are treated as unquantified properties that a system either does or does not possess. As Brewer himself later observed [25], and as Abadi emphasized in his PACELC

reformulation [1], real distributed systems rarely experience pure partitions; they experience *latency*, of which a partition is merely the limiting case where latency grows without bound. Furthermore, in cyber-physical systems (CPS) one cares about concrete delays measured in milliseconds, not abstract possibilities. The CAL theorem [112, 109] addresses these gaps by *quantifying* consistency, availability, and latency as time intervals and exhibiting a simple algebraic relationship among them. In doing so, it replaces the “P” in CAP with “L” (latency), elevating the CAP phenomenon from folklore to an engineering principle suitable for rigorous design of distributed real-time systems.

Model and definitions. The CAL theorem is developed over a trace-based model due to Schwartz and Mattern [166], in which a distributed execution consists of N sequential processes, each of which produces a stream of tagged events. Every event carries a *tag* $g \in \mathbb{G}$ drawn from a totally ordered set of tags, together with a *timestamp* $\mathcal{T}(g) \in \mathbb{T}$ obtained from a monotonically nondecreasing map $\mathcal{T}: \mathbb{G} \rightarrow \mathbb{T}$. Timestamps are drawn from the same domain as readings of local physical clocks, which allows logical and physical time to be compared, while remaining firmly a *logical* property of the program [109]. Events come in four kinds: *write* and *read* events that update and query a local replica of a shared variable x ; *send* events that launch an update across the network; and *receive* events that install an update received from the network. Tags are monotone across the network: if a send event with tag g_j on process j yields a receive event with tag g_i on process i , then $g_i \geq g_j$.

Given this model, the theorem quantifies four key notions. First, *inconsistency* $\bar{C}_{ij}$ between processes j and i for a shared variable x is the worst-case *logical-time* skew between the write of x on j and the corresponding receive on i :

$$\bar{C}_{ij} = \max(\mathcal{T}(g_i) - \mathcal{T}(g_j)),$$

where the maximization ranges over all write events on process j , with g_j denoting the tag of the write event and g_i the tag of the corresponding receive event on process i . Strong consistency corresponds to $\bar{C}_{ij} = 0$, and finite $\bar{C}_{ij}$ corresponds to a *quantified eventual consistency* [109]. Second, *unavailability* $\bar{A}_i$ at process i is the worst-case *physical-time* delay between the timestamp of a read event and the physical time at which that event is actually processed:

$$\bar{A}_i = \max(T_i - \mathcal{T}(g_i)),$$

where T_i is the physical time at which the read event is processed and g_i is its tag, and the maximization ranges over all read events on process i that are triggered by user requests. For CPS applications in which timestamps record the moment of sensor acquisition, $\bar{A}_i$ is precisely the sensor-to-actuator delay.

Third, the *processing offset* O_i at process i is defined as

$$O_i = \max(T_i - \mathcal{T}(g_i)),$$

where the maximization ranges over all write events on process i that are triggered by local external inputs. Structurally, the processing offset resembles the unavailability, but the

former concerns *write* events whereas the latter concerns *read* events. To maintain causal consistency, a process must ensure that all events with earlier tags have been processed before handling a new write event triggered by an external input; the processing offset measures the resulting physical-time delay. By definition, $O_i \geq 0$, and in many designs, $O_i = 0$, but when a process must wait for network messages with earlier tags before it can safely handle a locally triggered event, O_i becomes strictly positive.

Fourth, the *apparent latency* $\mathcal{L}_{ij}$ from process j to process i is the worst-case gap between the timestamp of a write on j and the physical time of the corresponding receive on i ,

$$\mathcal{L}_{ij} = \max(T_i - \mathcal{T}(g_j)) = O_j + X_{ij} + L_{ij} + E_{ij},$$

where $\mathcal{T}(g_j)$ is the timestamp of the write event on process j and T_i is the physical time of the corresponding receive event on process i . The decomposition lumps together four components that are typically hard to separate in practice but easy to measure together: the processing offset O_j at the sender, the execution overhead X_{ij} of sending, the raw network latency L_{ij} , and the clock synchronization error E_{ij} [109]. Because E_{ij} may be negative, apparent latency itself may be negative, which exposes the fact that in the absence of a shared clock the perceived order of events is itself logical, not physical.

The theorem. With these definitions, the CAL theorem states that, for any trace,

$$\bar{A}_i = \max\left(O_i, \max_{j \in N} (\mathcal{L}_{ij} - \bar{C}_{ij})\right), \quad (2.1)$$

i.e., the unavailability at node i is dominated either by the local processing offset or by the largest incoming apparent latency, *minus* the inconsistency that the designer is willing to tolerate along that link [112, 109]. The proof is immediate from the definitions: unavailability is determined by whichever update event (write or receive) has the worst gap between its timestamp and the physical time of its processing, and for a receive event the gap is exactly $\mathcal{L}_{ij}$ reduced by whatever logical delay has been inserted to widen the consistency tolerance. Collecting the pairwise quantities into an $N \times N$ matrix $\Gamma_{ij} = \mathcal{L}_{ij} - \bar{C}_{ij} - O_j = X_{ij} + L_{ij} + E_{ij} - \bar{C}_{ij}$ and stacking the $\bar{A}_i$ and O_i into vectors $\mathbf{A}$ and $\mathbf{O}$, the CAL theorem (2.1) becomes a *linear* equation in the max-plus (tropical) algebra [12]:

$$\mathbf{A} = (\mathbf{I} \oplus \Gamma) \mathbf{O}. \quad (2.2)$$

The processing offsets themselves satisfy $\mathbf{O} = \mathbf{Z} \oplus \Gamma \mathbf{O}$, whose least solution, when every cycle of Γ has strictly negative weight, is given by the Kleene star

$$\mathbf{O} = \Gamma^* \mathbf{Z} = (\mathbf{I} \oplus \Gamma \oplus \dots \oplus \Gamma^{N-1}) \mathbf{Z}. \quad (2.3)$$

The cycle-weight condition has a striking interpretation: along every feedback loop in the communication topology, the designer must tolerate *some* inconsistency, or else no finite processing offset exists and the system must wait forever, suggesting the ultimate loss of availability [109].

Significance and relation to CAP. The CAL theorem (2.1) subsumes the CAP theorem as a limiting case: as the apparent latency $\mathcal{L}_{ij}$ grows without bound (the formal signature of a network partition), at least one of $\bar{C}_{ij}$ or $\bar{A}_i$ must also grow without bound. Thus CAP’s impossibility is recovered as an edge case of a continuous trade-off curve, and designers gain a quantitative knob: for every millisecond of inconsistency one is willing to tolerate on a link, one gains a millisecond of availability on the downstream node. Unlike earlier quantification attempts that defined availability and consistency as fractions of satisfied accesses or out-of-order writes (and for which computing the trade-off is NP-hard [193]), the CAL formulation is linear in max-plus and admits closed-form solutions over arbitrary heterogeneous topologies with imperfect clocks [112].

Practical implications. Because consistency, availability, and latency are all expressed in the same units (time), the CAL theorem can be used in two complementary directions by a system designer [109]. From the top down, given application-level availability requirements (e.g., deadlines on actuation) and consistency requirements (e.g., tolerated logical skew between replicas), one can derive the *network latency budget* $X_{ij} + L_{ij} + E_{ij}$ that any deployment must respect, guiding placement decisions between end devices, edge computers, and the cloud, and informing the choice of networking fabric (e.g., time-sensitive networking on a factory floor versus a WAN). From the bottom up, given measured or assumed link latencies and clock-synchronization errors, one can compute the achievable unavailability and decide at which nodes deadlines can be met and at which nodes fault handlers must be installed. Two CPS examples in [109] make the application-specific nature of the trade-off concrete: a four-way autonomous intersection must prioritize consistency (vehicles stop rather than enter an intersection under disagreement), whereas an ADAS brake-pedal override must prioritize availability (the vehicle brakes immediately rather than waiting for a cloud-computed vision update). The theorem is integrated with the Lingua Franca coordination language [128, 124], in which **after** delays explicitly declare tolerated inconsistency $\bar{C}_{ij}$ and **deadline** clauses explicitly declare availability bounds, and in which both centralized [96] and decentralized [195, 40] coordinators realize arbitrary points on the CAL trade-off curve. In this way, the CAL theorem turns a long-standing piece of distributed-systems folklore into a design-time calculus for distributed real-time and cyber-physical systems.

Max-Plus Algebra

The *max-plus* (or *tropical*) semiring $\mathbb{R}_{\max} = (\mathbb{R} \cup \{-\infty\}, \oplus, \otimes)$ replaces the usual addition and multiplication with $a \oplus b := \max(a, b)$ and $a \otimes b := a + b$, taking $\varepsilon := -\infty$ as the additive identity ($a \oplus \varepsilon = a$) and $e := 0$ as the multiplicative identity ($a \otimes e = a$) [12, 67]. Although these operations are elementary, the resulting algebra turns synchronization constraints, such as waiting for the latest of several predecessors and then adding a processing delay, into a linear formalism, making systems of max-+ recurrences amenable to the same style of analysis that classical linear algebra provides for additive systems.

Max-plus matrix multiplication. For matrices $A \in \mathbb{R}_{\max}^{m \times p}$ and $B \in \mathbb{R}_{\max}^{p \times n}$, their max-plus product $C = A \otimes B$ is defined entry-wise as

$$C_{ij} = \bigoplus_{k=1}^p A_{ik} \otimes B_{kj} = \max_k (A_{ik} + B_{kj}).$$

Repeated matrix powers $A^k := A \otimes A^{k-1}$ (with $A^0 = I$, the max-plus identity matrix whose diagonal entries are 0 and off-diagonal entries are ε) encode the weight of the heaviest (longest-delay) path of exactly k edges in the weighted directed graph associated with A .

Kleene star. The *Kleene star* of an $n \times n$ matrix A is defined as

$$A^* = I \oplus A \oplus A^2 \oplus \dots = \bigoplus_{k=0}^{\infty} A^k.$$

Entry $(A^*)_{ij}$ equals the weight of the *longest* (critical) path from node j to node i , over paths of any length. The series converges to a finite matrix if and only if every directed cycle in the graph of A has strictly negative total weight; in that case the finite sum $A^* = I \oplus A \oplus \dots \oplus A^{n-1}$ suffices [12]. When a cycle has non-negative weight, repeated traversal can increase the path weight without bound, indicating that no finite schedule exists.

Max-plus subtraction $\oslash$. In the max-plus algebra, since $\otimes$ corresponds to ordinary addition ($a \otimes b = a + b$), its dual $\oslash$ corresponds to ordinary subtraction: $a \oslash b = a - b$. For vectors, $\oslash$ operates element-wise: $(u \oslash v)_i = u_i - v_i$.

Role in this chapter. Throughout this chapter we model the firing times of reactions as a max-plus linear recurrence of the form $X(k') = M \otimes X(k) \oplus N \otimes \bar{t}(k')$ (Section 2.5), and the CAL relation $\mathbf{A} = (\mathbf{I} \oplus \Gamma) \mathbf{O}$ with least solution $\mathbf{O} = \Gamma^* \mathbf{Z}$ (Section 2.2). The Kleene star Γ^* captures the cumulative propagation of delays through all paths in the communication topology, and the max-plus eigenvalue of M identifies the system's critical-path latency (Section 2.4). These constructions allow us to derive closed-form expressions for processing offsets and throughput bounds that would otherwise require simulation or ad-hoc reasoning.

2.3 System-Level Timing Analysis Framework

This section presents the core analytical framework. Whereas the original CAL theorem reasons about read and write events on replicated shared variables, the framework developed here reasons about the invocation of general *sequential procedures*, each of which may internally perform arbitrary reads, writes, or computations. This shift in granularity, from individual data-access events to procedure invocations, makes the framework applicable to a broad class of systems beyond replicated data stores, including cyber-physical control loops,

signal-processing pipelines, and any distributed program whose components are triggered reactively.

The framework applies to any system whose model of computation is based on *logical time* [125]: events carry logical timestamps drawn from the same domain as physical clock readings, and the runtime guarantees that procedures are invoked in logical-time order. The only requirement is that logical time values be expressed in the same units as physical time; the choice of units is otherwise arbitrary. This logical-time foundation ensures that the system's behavior is deterministic: the same inputs always produce the same outputs in the same logical order. The physical timing of procedure invocations, however, depends on execution times, network latencies, and coordination overhead. It is precisely this gap between deterministic logical behavior and variable physical timing that the framework captures algebraically.

We state the central result as Proposition 2.1, define its constituent matrices, and then show in Sections 2.4–2.6 how it is instantiated on concrete programs.

Setup. Consider a distributed system with n sequential procedures deployed across one or more *stateful containers*, i.e., nodes that maintain local state and execute their procedures sequentially. Each procedure is reactively triggered: it begins executing only when all of its triggering conditions are satisfied. We index the successive logical tags at which at least one procedure is invoked by $k = 1, 2, \dots$, writing $t(k)$ for the logical timestamp of the k -th tag. The state of the system at the k -th tag is captured by the vector

$$X(k) = \begin{bmatrix} x_1(k) \\ x_2(k) \\ \vdots \\ x_n(k) \end{bmatrix} \in \mathbb{R}_{\max}^n,$$

where $x_i(k)$ denotes the *exact physical time* at which procedure i can begin executing at the k -th tag.

The timing vector $E(k)$. The execution times of all procedures at the k -th tag are collected into the vector

$$E(k) = \begin{bmatrix} e_1(k) \\ e_2(k) \\ \vdots \\ e_n(k) \end{bmatrix} \in \mathbb{R}^n,$$

where $e_i(k)$ is the execution time of procedure i at the k -th tag.¹ If procedure i is not invoked at the k -th tag, then $e_i(k) = 0$ (the multiplicative identity in the max-plus algebra), so that

¹We use real numbers here to make the formalism algebraic, but integers and other domains could work as well, as long as logical and physical timelines share the same unit.

the procedure imposes no delay on its successors. While we present $e_i(k)$ as the execution time of a single procedure for simplicity, the symbol can be generalized to subsume other physical delays along the path from one procedure to another, including network latencies and clock synchronization errors. In this broader reading, $e_i(k)$ plays a role analogous to the *apparent latency* $\mathcal{L}_{ij}$ in the original CAL theorem [109], which similarly aggregates multiple sources of delay into a single measurable quantity. The entries of $E(k)$ appear throughout the matrices defined below: they form the edge weights in $\Gamma(k)$, the barrier delays in $B(k)$, and thereby determine the system's physical timing behavior at each tag.

Proposition 2.1. *For successive tags k and $k' = k + 1$, the state of the system satisfies*

$$X(k') = \Gamma^*(k') \left(B(k)X(k) \oplus \bar{t}(k') \right),$$

where matrices $\Gamma^*(k')$, $B(k)$, and the vector $\bar{t}(k')$ are subsequently defined.

Indexing convention. Throughout this chapter, all matrices use the convention that entry (i, j) records the constraint that procedure j imposes on procedure i , i.e., “ j constrains i .” This convention follows from the standard max-plus state equation $X(k') = A \otimes X(k)$ [12], where the state vector X is a column vector and the i -th component expands as $x_i(k') = \bigoplus_j A_{ij} \otimes x_j(k)$. Consequently, row i of a matrix collects all the constraints that other procedures impose *on* procedure i , while column j collects all the constraints that procedure j imposes *on others*.

The intra-tag dependency matrix $\Gamma(k')$. Within a single tag k' , some procedures may depend on others: a procedure may need the output of another procedure that executes at the same tag before it can begin. Such dependencies arise from priority ordering among procedures within the same container, or from zero-delay connections between containers. The matrix

$$\Gamma(k') = \begin{bmatrix} \Gamma_{11}(k') & \Gamma_{12}(k') & \cdots & \Gamma_{1n}(k') \\ \Gamma_{21}(k') & \Gamma_{22}(k') & \cdots & \Gamma_{2n}(k') \\ \vdots & \vdots & \ddots & \vdots \\ \Gamma_{n1}(k') & \Gamma_{n2}(k') & \cdots & \Gamma_{nn}(k') \end{bmatrix} \in \mathbb{R}_{\max}^{n \times n}$$

encodes these intra-tag dependencies. Its entry $\Gamma_{ij}(k')$ equals the execution time $e_j(k')$ if procedure i must wait for procedure j to finish at the same tag k' before it can begin, and equals $\varepsilon = -\infty$ otherwise. Diagonal entries satisfy $\Gamma_{ii}(k') = \varepsilon$, since a procedure does not depend on itself within a single tag.

The dependency closure $\Gamma^*(k')$. The Kleene star

$$\Gamma^*(k') = \begin{bmatrix} \Gamma_{11}^*(k') & \Gamma_{12}^*(k') & \cdots & \Gamma_{1n}^*(k') \\ \Gamma_{21}^*(k') & \Gamma_{22}^*(k') & \cdots & \Gamma_{2n}^*(k') \\ \vdots & \vdots & \ddots & \vdots \\ \Gamma_{n1}^*(k') & \Gamma_{n2}^*(k') & \cdots & \Gamma_{nn}^*(k') \end{bmatrix} \in \mathbb{R}_{\max}^{n \times n}$$

is the transitive closure of $\Gamma(k')$, defined by the max-plus Kleene series

$$\Gamma^*(k') = I \oplus \Gamma(k') \oplus \Gamma^2(k') \oplus \dots$$

For the intra-tag dependency matrices considered here, this infinite series reduces to the finite expression

$$\Gamma^*(k') = I \oplus \Gamma(k') \oplus \Gamma^2(k') \oplus \dots \oplus \Gamma^{n-1}(k').$$

This follows from Theorem 3.20 of Baccelli et al. [12]: if the graph of a max-plus matrix has no circuit with positive weight, then its Kleene star is obtained by considering paths of length at most $n - 1$. Entry $\Gamma_{ij}^*(k')$ gives the weight of the longest dependency chain from procedure j to procedure i within the same tag, accounting for all intermediate execution times. Diagonal entries satisfy $\Gamma_{ii}^*(k') = 0$ (the max-plus multiplicative identity), reflecting the fact that a procedure's dependency on itself adds no extra delay. The matrix $\Gamma^*(k')$ depends on the execution times at tag k' because the weight of each dependency edge is the execution time of the upstream procedure at that tag. The series is guaranteed to converge (i.e., the finite sum suffices) whenever the intra-tag dependency graph is acyclic, which is the case for well-formed programs in which no circular same-tag dependencies exist.²

The cross-tag dependency matrix $B(k)$. Before a container can advance from tag k to tag k' , *all* procedures within that container at tag k must have finished executing.³ The matrix $B(k) \in \mathbb{R}_{\max}^{n \times n}$ encodes this constraint. Its entry $B_{ij}(k)$ equals the execution time $e_j(k)$ if procedure j at tag k must complete before procedure i can begin at tag k' , and equals $\varepsilon = -\infty$ otherwise.

In a base case, where every procedure can advance time independently of others (i.e., every container has only one procedure), the B matrix is diagonal:

$$B_{base}(k) = \begin{bmatrix} e_1(k) & \varepsilon & \cdots & \varepsilon \\ \varepsilon & e_2(k) & \cdots & \varepsilon \\ \vdots & \vdots & \ddots & \vdots \\ \varepsilon & \varepsilon & \cdots & e_N(k) \end{bmatrix},$$

where N is the total number of procedures in the system. In distributed logical time systems, it is more common for a group of procedures to advance time together in the same container.

²Later, when we map this construction to LINGUA FRANCA (LF), a valid LF program should have no cyclic dependencies; hence the condition that there be no circuit with positive weight holds. If an LF program has cyclic dependencies, the LF toolchain detects them, and the matrix representation exposes the same problem: $\Gamma^*(k')$ has circuits with positive weight, so Theorem 3.20 no longer applies.

³This is a side note on the notion of “containers” in the system model. In theory, the system model does not necessarily need “containers” because the model primarily relies on *dependencies*. Containers are a prior for dependencies. Once dependencies are established, containers are no longer needed. In our case, we are distinguishing between two types of dependencies: intra-tag and cross-tag. For cross-tag dependencies, the container abstraction is more intuitive because it expresses the fact that procedures in the same container advance logical time together, hence I decided to keep containers in the model.

Consequently, the general $B(k)$ is *block diagonal*:

$$B(k) = \begin{bmatrix} B_1(k) & \varepsilon & \cdots & \varepsilon \\ \varepsilon & B_2(k) & \cdots & \varepsilon \\ \vdots & \vdots & \ddots & \vdots \\ \varepsilon & \varepsilon & \cdots & B_m(k) \end{bmatrix},$$

where m is the total number of containers, and each block $B_c(k)$ is a square submatrix whose rows and columns are indexed by the procedures of container c , and all off-diagonal blocks are filled with ε (no cross-container barriers). Every entry of the submatrix $B_c(k)$ is equal to the corresponding $e_j(k)$: each procedure in container c waits for every procedure in the same container to finish. Assuming n procedures in container c , the submatrix $B_c(k)$ takes the following form:

$$B_c(k) = \begin{bmatrix} | & | & & \cdots & | \\ e_1(k) & e_2(k) & \cdots & e_n(k) \\ | & | & & \cdots & | \end{bmatrix},$$

where the vertical bars in each column indicate that every entry equals the execution time shown in the middle row.

In the extreme case, *i.e.*, a single-process execution with *global* barrier synchronization, where the only federate cannot advance logical time until every procedure at the current tag has completed, the block structure disappears: assuming again a total of N procedures, every column of $B(k)$ is fully populated with $e_j(k)$, yielding

$$B(k) = \begin{bmatrix} | & | & & \cdots & | \\ e_1(k) & e_2(k) & \cdots & e_N(k) \\ | & | & & \cdots & | \end{bmatrix}.$$

The logical timestamp vector $\bar{t}(k)$. The vector

$$\bar{t}(k) = \begin{bmatrix} t(k) \\ t(k) \\ \vdots \\ t(k) \end{bmatrix} \in \mathbb{R}_{\max}^n$$

broadcasts the logical timestamp of the k -th tag to every procedure.⁴ The term $\bar{t}(k')$ in Proposition 2.1 enforces the *physical realizability constraint*: no procedure may begin ex-

⁴An earlier version of the proposition uses a scalar $t(k')$, rather than a vector, written in the form $X(k') = M(k, k') \otimes X(k) \oplus N(k') \cdot t(k')$, where $M(k, k')$ is the combined state-transition matrix and $N(k')$

executing at tag k' before the physical clock reaches the logical time $t(k')$. This constraint anchors the recurrence to physical time and ensures that procedures whose logical time lies in the future cannot be executed prematurely.

Interpretation. Proposition 2.1 can be read as a two-phase recurrence. In the first phase, the term $B(k)X(k)$ propagates the consequences of tag k : each procedure's earliest start time at tag k' is constrained by the completion times of all procedures that must finish before the container allows logical time advancement. This result is then combined ($\oplus$) with the next logical timestamp $\bar{t}(k')$, which enforces the real-time constraint that an event can only be processed when the current physical time surpasses its logical timestamp. In the second phase, multiplication by $\Gamma^*(k')$ propagates same-tag dependencies: if procedure i depends on procedure j at the same tag, then i 's start time is delayed by j 's execution time. The Kleene star ensures that transitive chains of such dependencies are fully resolved.

A slightly different interpretation arises after distributing $\Gamma^*(k')$ into the parentheses, which rewrites Proposition 2.1 as

$$\begin{aligned} X(k') &= \Gamma^*(k')B(k)X(k) \oplus \Gamma^*(k')\bar{t}(k') \\ &= M(k', k)X(k) \oplus \Gamma^*(k')\bar{t}(k'), \end{aligned}$$

where $M(k', k) = \Gamma^*(k')B(k)$, defined as the *system matrix*, combines both the intra-tag ($\Gamma^*(k')$) and cross-tag ($B(k)$) dependencies. $M(k', k)X(k)$ represents the earliest k' -th firing times resulting from processing the previous k -th tag, while $\Gamma^*(k')\bar{t}(k')$ represents the earliest k' -th firing times resulting from new events at the k' -th tag. The larger of the two becomes the next $X(k')$.

There are five key assumptions on the system model underlying Proposition 2.1:

1. Procedures are invoked at well-defined logical tags, with acyclic intra-tag and cross-tag dependencies (mathematically represented as $\Gamma^*(k')$ and $B(k)$).
2. Execution times can be different at each logical tag ($E(k)$ is a function over tag index $k \in \mathbb{N}$, which further turns dependency matrices into functions of k).
3. A logical tag is processed if and only if the current physical time surpasses the logical timestamp (intuitively, logical time “chases” physical time), represented by $\bar{t}(k')$.

is a column vector. The two formulations are equivalent. First, $M(k, k') = \Gamma^*(k') \otimes B(k)$ by construction. Second, the column-vector $N(k')$ times the scalar $t(k')$ equals $\Gamma^*(k') \otimes \bar{t}(k')$: the i -th entry of $\Gamma^*(k') \otimes \bar{t}(k')$ is $\max_j (\Gamma_{ij}^*(k') + t(k')) = \max_j \Gamma_{ij}^*(k') + t(k')$ (since $t(k')$ is a scalar), and the row-max $\max_j \Gamma_{ij}^*(k')$ is exactly the i -th entry of $N(k')$ (this relies on execution times being non-negative, which always holds). Combining these, $M \otimes X(k) \oplus N \cdot t(k') = \Gamma^* \otimes B \otimes X(k) \oplus \Gamma^* \otimes \bar{t}(k')$, and by distributivity of $\otimes$ over $\oplus$ this equals $\Gamma^* \otimes (B \otimes X(k) \oplus \bar{t}(k'))$, which is Proposition 2.1. The vector form $\bar{t}(k')$ is cleaner because it makes the role of the logical timestamp symmetric with the other operands and avoids mixing scalar and matrix multiplication in the same equation.

4. Delayed connections satisfy the LET discipline: each logical delay bounds the maximum accumulated execution time along zero-delay upstream paths feeding that connection, stopping at any earlier delayed connection; for example, in $A \xrightarrow{0} B \xrightarrow{0} C \xrightarrow{\text{delay1}} D$, then delay1 must satisfy $\text{delay1} \geq e_A + e_B + e_C$, but if instead $A \xrightarrow{0} B \xrightarrow{\text{delay2}} C \xrightarrow{\text{delay1}} D$, then $\text{delay1} \geq e_C$ only, since $\text{delay2} \geq e_A + e_B$.
5. There are unbounded (but finite) computational resources such that procedures can be scheduled to run in parallel whenever possible.

Removing Assumption 3 reduces Proposition 2.1 to classical results in Baccelli et al. [12] and Lee et al. [102], where firing times at the k' -th tag depend strictly on processing the previous k -th tag and not on the new timestamp at the k' -th tag. Assumption 4 is the key to preserve the simplicity of Proposition 2.1. We later show that removing this assumption turns the simple recurrence, where tag index k' only depends on the preceding index k , into a convolution, where k' depends on all prior indices $(0, 1, \dots, k)$.

The subsequent case studies (Sections 2.4–2.6) demonstrate how to construct $\Gamma(k')$, $B(k)$, and $\bar{t}(k')$ for concrete system topologies and how to use the resulting recurrence for engineering analysis.

Exploring Consistency-availability Trade-offs

The unavailability at the k' -th tag, where $k' = k + 1$, is defined as the difference between when procedures can physically fire and the logical time:

$$\bar{A}(k') = X(k') \oslash \bar{t}(k') = \Gamma^*(k') \otimes \left(B(k) \otimes X(k) \oslash \bar{t}(k) \oplus \vec{0} \right), \quad (2.4)$$

where $\oslash$ denotes element-wise subtraction (the dual of $\otimes$ in the max-plus algebra) and $\vec{0}$ is the zero vector.

To see where the consistency-availability trade-off emerges, consider a write event w_1 associated with a logical delay $\bar{C}_1$, sent at the k_1 -th tag and received at the k' -th tag, so that $t(k_1) + \bar{C}_1 = t(k')$. Substituting:

$$\bar{A}(k') = \Gamma^*(k') \otimes \left(B(k) \otimes X(k) \oslash (\bar{t}(k_1) + \bar{C}_1) \oplus \vec{0} \right).$$

As $\bar{C}_1$ increases (sacrificing consistency), $\bar{A}(k')$ decreases (improving availability), until $\vec{0}$ dominates. Strong consistency corresponds to $\bar{C}_1 = 0$, which maximizes unavailability.

When a procedure has multiple triggers with distinct logical delays $\bar{C}_1$ and $\bar{C}_2$, two scenarios arise. If the corresponding write events are received at the same tag, the unavailability is expressed with a single equation as above. If they are received at different tags $t(p')$ and $t(q')$, the unavailability must be expressed separately for each tag:

$$\begin{aligned} \bar{A}(p') &= \Gamma^*(p') \otimes \left(B(p) \otimes X(p) \oslash (\bar{t}(k_1) + \bar{C}_1) \oplus \vec{0} \right), \\ \bar{A}(q') &= \Gamma^*(q') \otimes \left(B(q) \otimes X(q) \oslash (\bar{t}(k_2) + \bar{C}_2) \oplus \vec{0} \right). \end{aligned}$$

In this case, the logical timeline $t(k)$ must be treated as an external input that defines $t(p') = t(k_1) + \bar{C}_1$ and $t(q') = t(k_2) + \bar{C}_2$.

Relation to the Original CAL Theorem

The factored form of Proposition 2.1 mirrors the structure of the original CAL theorem’s processing-offset equation $\mathbf{O} = \Gamma^* \mathbf{Z}$ (Equation 2.3), where Γ^* encodes same-tag propagation and $\mathbf{Z}$ encodes the exogenous inputs. The correspondence becomes especially clear in the unavailability equation (Equation (2.4)):

$$\bar{A}(k') = \Gamma^*(k') \otimes \left(B(k) \otimes X(k) \otimes \bar{t}(k') \oplus \vec{0} \right).$$

In the original theorem, the processing offset $\mathbf{O}$ is a lag, the worst-case difference between the physical time at which a write event is processed and its logical timestamp. In Equation (2.4), the left-hand side $\bar{A}(k') = X(k') \otimes \bar{t}(k')$ is likewise a lag: the element-wise difference between physical firing times and logical timestamps at tag k' . On the right-hand side, $\Gamma^*(k')$ plays the same role as in the original theorem, propagating same-tag dependencies. The $\vec{0}$ term corresponds to $\mathbf{Z} = \vec{0}$, the case in the original theorem where every node has local physically timestamped inputs and therefore incurs no exogenous offset. The additional term $B(k) \otimes X(k) \otimes \bar{t}(k')$, which has no counterpart in the original theorem’s static equation, captures the dynamic carry-over of delays from the previous tag.

Beyond this structural parallel, two key differences distinguish Proposition 2.1 from the original theorem. First, the original theorem reasons about read and write events on shared variables, whereas Proposition 2.1 reasons about the invocation of general sequential procedures that may contain arbitrary reads and writes. Second, the original theorem operates on worst-case quantities aggregated over an entire execution, whereas Proposition 2.1 tracks *exact* physical firing times at every tag, accepting per-event execution times and latencies as input.

General Form without the LET Assumption

Assumption 4 is what makes Proposition 2.1 a first-order recurrence over logical tags (i.e., the state at tag k' depends only on the immediately preceding tag $k' - 1$, rather than on the full history of earlier tags). It requires every logical delay to be large enough to cover the accumulated physical execution time along the upstream zero-delay path that produces the delayed event. Under that condition, an event sent at logical time $t(k^-)$ and delivered with logical delay d is guaranteed to be physically available by the time the destination tag $t(k') = t(k^-) + d$ is processed. The delayed connection can therefore be represented purely by the logical timestamp term $\bar{t}(k')$; no explicit dependency on $X(k^-)$ is needed.

If this LET assumption is removed, a delayed connection can carry memory from older tags. For example, suppose reaction A sends to reaction B through a connection with logical delay 10s, while A is triggered every one second. As shown in Figure 2.1, if A takes 20s to

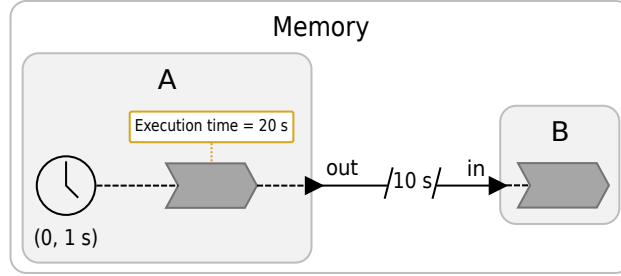


Figure 2.1: A delayed connection can introduce memory across logical tags when its logical delay does not cover the upstream execution time. The downstream reaction at tag k' must wait both for the logical tag to appear and for the upstream reaction at the earlier tag to complete.

execute, then A 's execution time exceeds the logical delay, and the event with destination logical time $t(k^-) + 10\text{s}$ is not physically produced by that logical time. The start time of B at the destination tag is then constrained not only by the appearance of the logical tag, but also by the completion of A at the sending tag:

$$x_B(k') \geq \max(t(k'), x_A(k^-) + e_A(k^-)).$$

Thus the delayed connection is no longer hidden inside $\bar{t}(k')$. It contributes an explicit dependency from an older state vector.

The corresponding general form replaces the first-order term $\Gamma^*(k')B(k)X(k)$ with a max-plus convolution over past tags. This convolutional structure has its origin in network calculus, where min-plus and max-plus convolutions are used to model accumulated timing constraints in communication and real-time systems [41, 183]:

$$X(k') = \bigoplus_{m=1}^{k'} D_m(k') \otimes X(k' - m) \oplus \Gamma^*(k') \otimes \bar{t}(k'). \quad (2.5)$$

Here $D_m(k') \in \mathbb{R}_{\max}^{n \times n}$ is a *memory matrix*. Entry (i, j) of $D_m(k')$ records the timing constraint that reaction j at tag $k' - m$ imposes on reaction i at tag k' , after same-tag dependencies at k' have been propagated. The current framework is recovered as the special case in which only the immediately preceding tag contributes:

$$\begin{aligned} D_1(k') &= \Gamma^*(k') \otimes B(k' - 1), \\ D_2(k') &= \mathcal{E}, \\ D_3(k') &= \mathcal{E}, \\ &\vdots \\ D_m(k') &= \mathcal{E} \quad \text{for all } m > 1, \end{aligned}$$

where $\mathcal{E}$ is the all- ε matrix.

When non-LET delayed connections are present, additional memory matrices become nontrivial. For the two-reaction system in Figure 2.1, ordered as (A, B) , the delayed edge from A to B whose source tag is eleven indices before the destination tag can be represented, under the start-time convention used in this chapter, by a memory matrix with the form

$$\begin{aligned} D_1(k') &= \Gamma^*(k') \otimes B(k' - 1), \\ D_2(k') &= \mathcal{E}, \\ D_3(k') &= \mathcal{E}, \\ &\vdots \\ D_{10}(k') &= \mathcal{E}, \\ D_{11}(k') &= \begin{matrix} & A & B \\ A & \begin{bmatrix} \varepsilon & \varepsilon \end{bmatrix} \\ B & \begin{bmatrix} e_A(k' - 11) & \varepsilon \end{bmatrix} \end{matrix}, \\ D_{12}(k') &= \mathcal{E}, \\ &\vdots \end{aligned}$$

The off-diagonal entry in row B , column A says that B at tag k' must wait for A at tag $k' - 11$ to complete. The subscript in D_{11} is an index offset determined by the fact that A fires once per second and the connection has logical delay 10s; it is not related to the 20s execution time in the example. If the state vector were instead defined in terms of completion times, this entry would be 0, yielding the simpler matrix that records only the precedence relation.

Equation (2.5) therefore identifies exactly what Assumption 4 buys: it collapses all delayed timing dependencies into the logical timestamp term and leaves a first-order recurrence. Dropping the assumption preserves max-plus linearity, but changes the system from a one-step recurrence into a finite-memory convolution over the prior tags whose events may still be physically pending. For LF programs, this memory horizon is bounded because logical delays on connections are fixed. Nevertheless, the general memory-enabled form is less tractable because the analysis must explicitly carry the finite history induced by those delayed connections. Thus for the rest of the chapter, we keep Assumption 4 for the LF programs under analysis, which enables us to use the more tractable Proposition 2.1.

Mapping to the Reactor Model

The framework presented above is stated in terms of generic stateful containers and sequential procedures. We now instantiate it for the *reactor* model of computation [126, 123], which provides the concrete system model used in the case studies that follow.

A reactor program is a composition of *reactors*, each of which maintains local state variables and hosts one or more *reactions*, which are sequential procedures triggered by events

arriving at the reactor’s input ports, timers, or actions. Reactors communicate through *connections* that carry tagged events between ports. A reactor can also embed child reactors, which do not share state with the parent reactor except through ports. The model of computation is based on *logical time* [125]: every event carries a logical timestamp, and the runtime guarantees that reactions within each reactor are invoked in logical-time order. This logical-time foundation ensures deterministic behavior: the same inputs always produce the same outputs in the same logical order, regardless of the physical platform on which the program executes.

A reactor system can either be executed as a single process, or as a distributed *federation* where top-level reactors, known as *federates*, run in separate processes. Under federated execution, top-level federate processes coordinate over a network interface and collectively implement the reactor semantics. Federates can execute in parallel at any given tag but implement a barrier synchronization before advancing the tag [16]. Since federated execution generalizes the single-process execution⁵, we will map our generic system model to a reactor federation, as follows:

- Each *stateful container* corresponds to a reactor *federate*.
- Each *sequential procedure* corresponds to a *reaction*, whether in a federate itself or in one of its nested child reactors.
- *Intra-tag dependencies* (Γ) arise from two sources: *priority ordering* among reactions within the same reactor (reactions with lower priority must wait for higher-priority reactions to complete at the same tag), and *zero-delay connections* between reactors (a downstream reaction must wait for the upstream reaction to produce its output at the same tag).
- *Cross-tag barriers* (B) arise from the reactor’s sequential execution model: all reactions within a reactor at tag k must complete before any reaction in that reactor can begin at tag $k' = k + 1$.
- The *logical timestamp* $t(k)$ is the timestamp component of the LINGUA FRANCA tag at the k -th invocation, expressed in the same units as physical time (e.g., nanoseconds since epoch).

With this mapping established, the remainder of this chapter uses reactor-model terminology: *reactors* (also known as *federates*) for containers and *reactions* for procedures.

2.4 Case Study: Ensuring System Responsiveness

We illustrate Proposition 2.1 with a small but instructive example: a stock exchange consisting of two reactors, a **Client** and an **Exchange**, shown in Figure 2.2. The client receives

⁵The single-process execution can be thought of as a federation with a single federate.

buy/sell orders from a user through a physical action and forwards them to the exchange over a connection. Orders arrive asynchronously; the exchange could become overloaded if orders are submitted too frequently.

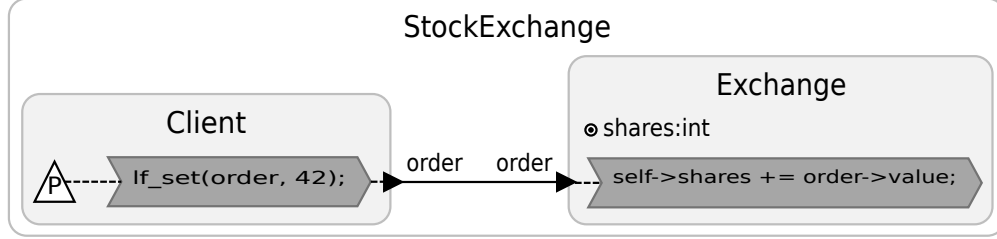


Figure 2.2: The world’s smallest stock exchange: a **Client** reactor receives asynchronous orders via a physical action and forwards them to an **Exchange** reactor.

The question we want to answer is: *What is the maximum order frequency the exchange can handle without becoming laggy?*

Modeling the System

We model the system behavior using a state vector

$$X(k) = \begin{bmatrix} x_C(k) \\ x_E(k) \end{bmatrix},$$

where k indexes logical tags and each $x_n(k)$ represents reaction n ’s exact *physical start time* at the k -th tag. For example, if the client’s handler fires for the first tag (say $t = 0$) at physical time $T = 5$ ms, then $x_C(1) = 5$ ms.

We also define:

- $E(k) = \begin{bmatrix} e_C(k) \\ e_E(k) \end{bmatrix}$: the reaction execution times at the k -th tag. If $e_n(k) = 0$, reaction n does not fire at the k -th tag.
- $t(k)$: the logical timestamp of the k -th tag.

Encoding System Constraints

Consider when each reaction can fire at the k' -th tag. For the client:

$$x_C(k') = \max(x_C(k) + e_C(k), t(k')). \quad (2.6)$$

The first term requires the client’s handler to finish its previous job at the k -th tag before processing the next order (barrier synchronization). The second term requires physical time to reach the k' -th tag’s logical time (the user submits the next order at $t(k')$).

For the exchange:

$$x_E(k') = \max(x_E(k) + e_E(k), x_C(k') + e_C(k')). \quad (2.7)$$

The first term is again barrier synchronization: the exchange must finish the previous job. The second term arises from the connection: the exchange must wait for the client to send the order at the k' -th tag.

Deriving the Max-Plus Linear Form

We eliminate the dependence of $x_E(k')$ on $x_C(k')$ by substituting Equation (2.6) into Equation (2.7) and distributing:

$$\begin{aligned} x_E(k') &= \max(x_E(k) + e_E(k), \max(x_C(k) + e_C(k), t(k')) + e_C(k')) \\ &= \max(x_E(k) + e_E(k), x_C(k) + e_C(k) + e_C(k'), t(k') + e_C(k')). \end{aligned}$$

Using the max-plus operators $\oplus := \max$ and $\otimes := +$ with $\epsilon := -\infty$, we write the matrix form directly in the factored shape of Proposition 2.1:

$$X(k') = \Gamma^*(k') \otimes (B(k) \otimes X(k) \oplus \bar{t}(k')), \quad (2.8)$$

where

$$\Gamma^*(k') = \begin{array}{c} C \quad E \\ E \left[\begin{array}{cc} 0 & \epsilon \\ e_C(k') & 0 \end{array} \right], \quad B(k) = \begin{array}{c} C \quad E \\ E \left[\begin{array}{cc} e_C(k) & \epsilon \\ \epsilon & e_E(k) \end{array} \right], \quad \bar{t}(k') = \begin{array}{c} C \\ E \left[\begin{array}{c} t(k') \\ t(k') \end{array} \right]. \end{array}$$

Here Γ^* captures the same-tag reaction dependency (the exchange depends on the client at the same tag), B captures barrier synchronization (each reaction must finish its prior invocation), and $\bar{t}(k')$ broadcasts the logical timestamp to enforce physical realizability. Expanding Equation (2.8) component-wise recovers the scalar equations derived above.

Using Eigenvalues to Identify System Critical Paths

The matrix $M(k, k') = \Gamma^*(k')B(k)$ is introduced earlier as the system matrix that combines both the system's intra-tag ($\Gamma^*(k')$) and cross-tag ($B(k)$) dependencies. To use this matrix for throughput analysis, we recall what an eigenvalue means in the max-plus algebra. For a max-plus matrix A , an eigenpair (λ, v) satisfies

$$A \otimes v = \lambda \otimes v,$$

where v is not the all- ϵ vector. Equivalently, each component satisfies $\max_j (A_{ij} + v_j) = \lambda + v_i$. The scalar λ is a per-step growth rate: if the recurrence repeatedly applies the same matrix,

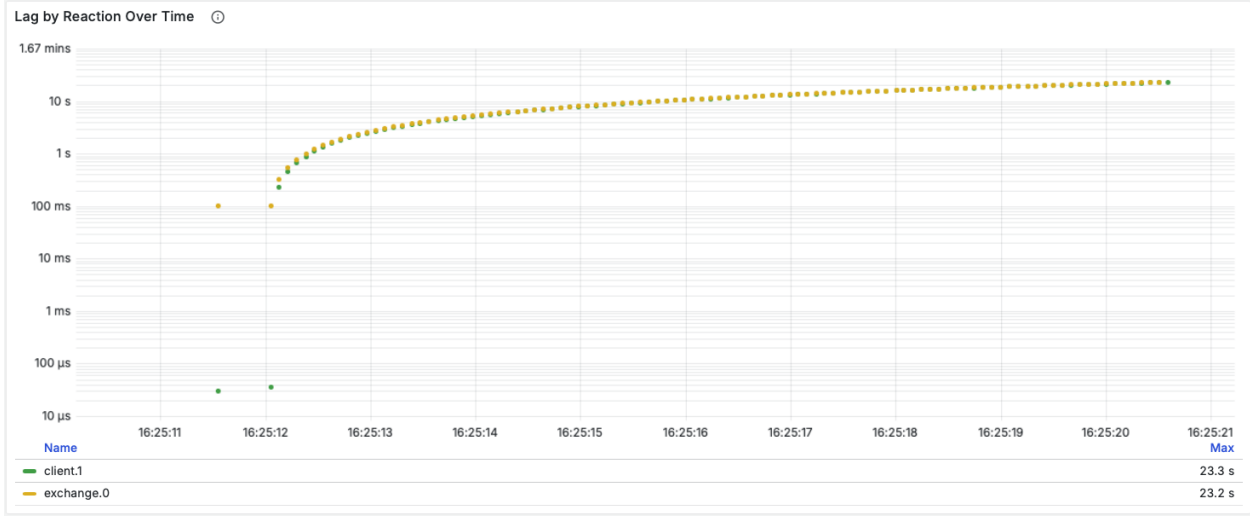


Figure 2.3: Lag measured under the original design, which grows without bound.

then λ is the amount by which the dominant firing times grow from one logical tag to the next.

The graph interpretation gives the critical-path intuition. Associate with A a weighted directed graph $G(A)$ whose nodes are the state-vector components and whose edge $j \rightarrow i$ has weight A_{ij} whenever $A_{ij} \neq \varepsilon$. A directed cycle in this graph represents a feedback path that must be traversed once per logical step, and its *cycle mean* is its total weight divided by its length. If $G(A)$ is strongly connected, equivalently if A is irreducible, then A has a unique max-plus eigenvalue, and this eigenvalue is exactly the maximum cycle mean [12, Theorem 3.23]. The eigenvector need not be unique, but the eigenvalue is.

If $G(A)$ is not strongly connected, equivalently if A is reducible, the situation is subtler. The strongly connected components (SCCs) of $G(A)$ each contribute cycle mean candidates, and every eigenvalue must be one of these cycle means [12, Remark 3.24]. However, not every SCC cycle mean is necessarily an eigenvalue: depending on how SCCs feed into one another and on their relative cycle means, a reducible matrix may have one eigenvalue or several. The maximum cycle mean is always an eigenvalue and is the largest eigenvalue; it is also the quantity that governs the asymptotic growth rate of the recurrence. Consequently, when we say that the eigenvalue of M determines the critical path below, we mean this dominant max-plus eigenvalue, or equivalently the maximum cycle mean of the dependency graph.

For a stationary system matrix, this dominant eigenvalue gives the longest recurring chain of execution that must complete between consecutive tags. If the time interval between consecutive orders is shorter than this dominant eigenvalue, the system will eventually fall behind.

For example, assume constant execution times $e_C(k) = 100 \text{ ms}$, $e_E(k) = 300 \text{ ms}$, $\forall k \in \mathbb{N}$,

then

$$M = \Gamma^* B = \begin{bmatrix} 0 & \epsilon \\ 100 & 0 \end{bmatrix} \begin{bmatrix} 100 & \epsilon \\ \epsilon & 300 \end{bmatrix} = \begin{bmatrix} 100 & \epsilon \\ 200 & 300 \end{bmatrix}$$

has max-plus eigenvalue $\lambda = 300$ ms. This matrix is reducible: its graph has one SCC for the client reaction with cycle mean 100 ms and one SCC for the exchange reaction with cycle mean 300 ms, with a dependency edge from the client SCC to the exchange SCC. Only the downstream exchange cycle mean appears as an eigenvalue in this case; the smaller client cycle mean is a candidate cycle mean but not an eigenvalue of the full matrix. The dominant eigenvalue is therefore the exchange’s recurring 300 ms path. This means that if the interval between consecutive orders is less than 300 ms, *i.e.*, $t(k') - t(k) < 300$ ms for all k , the exchange will eventually not be able to keep up. When M is stationary, as in this case, the eigenvalue analysis reveals a global fundamental limit.

Figure 2.3 shows the lag measured in real-time using the Xronos dashboard [190]. In this case, requests are sent with intervals less than 300 ms, which results in increasing lag. The demo program treats each key press on **Enter** as a “send order” command. The curve is the result of pressing and holding **Enter**.

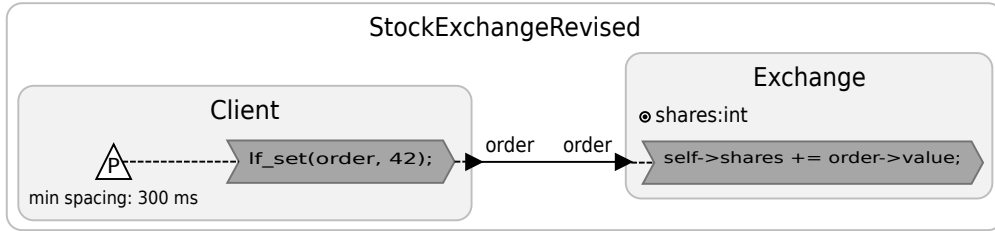


Figure 2.4: Revised stock exchange design with a minimum spacing derived from the max-plus eigenvalue, imposing a rate limit that prevents the exchange from falling behind.

This analysis directly suggests a design improvement: adding a minimum spacing of $\lambda = 300$ ms to the physical action on the client side, effectively imposing a rate limit derived from the system’s critical path. Figure 2.4 shows the revised design. Figure 2.5 shows the lag measured under the revised design, which shows bounded lag due to the added minimum spacing.

2.5 Case Study: Systems with Feedback Loops

This case study demonstrates how Proposition 2.1 can be applied to a concrete distributed system to derive execution policies, specifically, processing offsets that enable decentralized coordination. We introduce a running example called *Impossible Consistency*, walk through the modeling and algebraic derivation in detail, and show how the result directly yields the *maxwait* parameters [150] used in LINGUA FRANCA’s decentralized coordination.

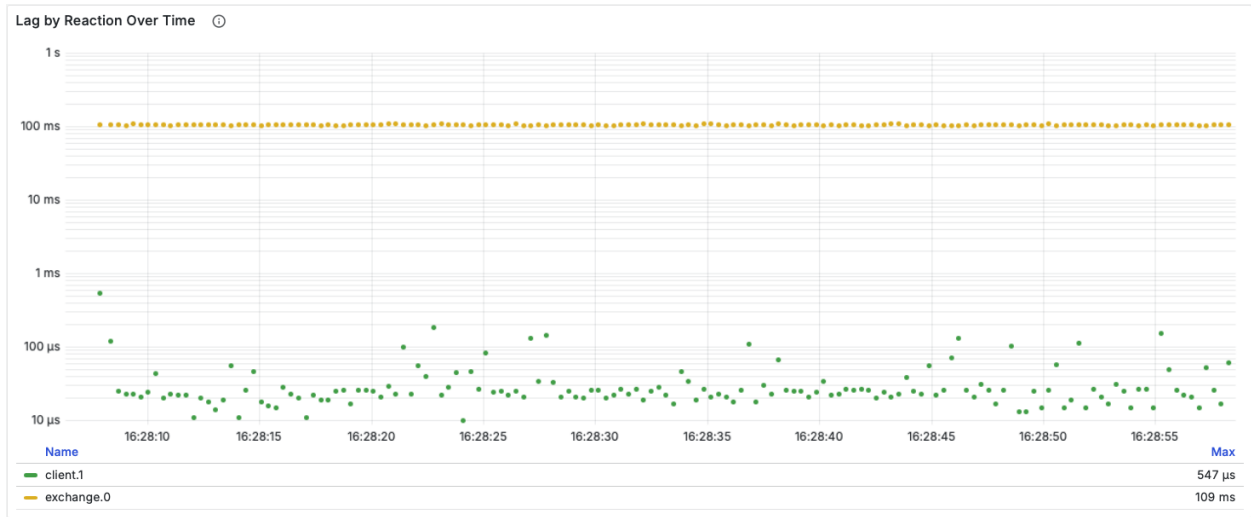


Figure 2.5: Lag measured under the revised design, which stays bounded.

Consider a simple distributed system consisting of two reactors, A and B , each with a physical action and two reactions. Both reactors receive asynchronous updates from the environment through their physical actions and are connected by zero-delay connections, requiring strong consistency across nodes. Figure 2.6 illustrates this setup.

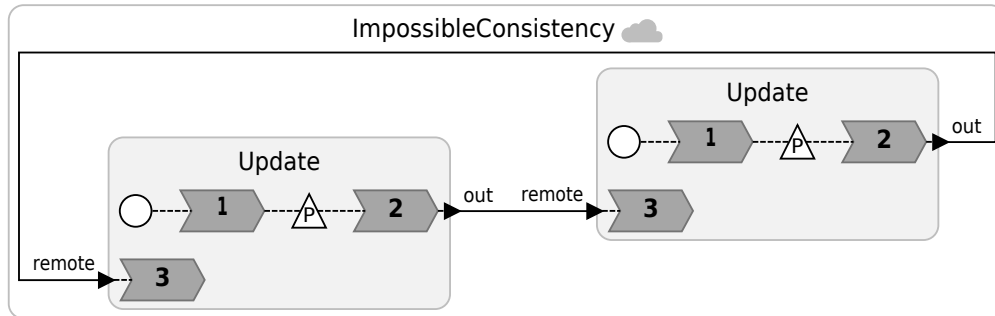


Figure 2.6: The Impossible Consistency example: a distributed system of two reactors receiving asynchronous updates while maintaining strong consistency.

The central question is: *How do we model the system's behavior over time?* In particular, we want to understand when each reaction can fire and, ultimately, what processing offsets are required for decentralized coordination.

For clarity, we simplify the diagram by removing the reactions triggered by startup, focusing on the core structure shown in Figure 2.7.

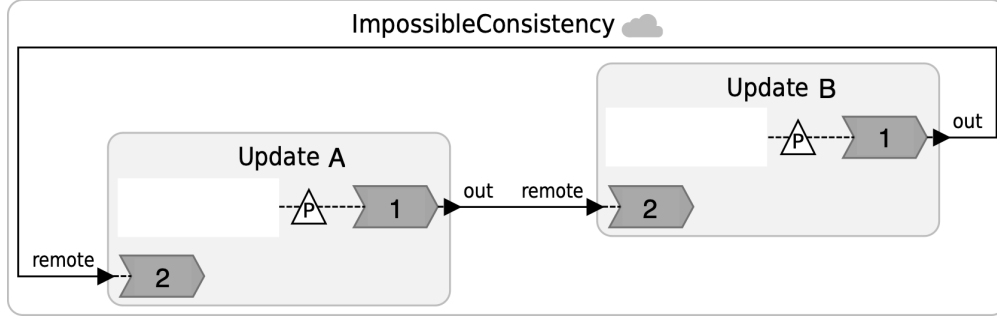


Figure 2.7: Simplified diagram of the Impossible Consistency example, with startup-triggered reactions removed.

Modeling Reaction Firing Times

We model the system behavior using a state vector

$$X(k) = \begin{bmatrix} x_{A1}(k) \\ x_{A2}(k) \\ x_{B1}(k) \\ x_{B2}(k) \end{bmatrix},$$

where k represents the k -th logical tag at which a reaction is invoked. Each $x_n(k)$ represents the *earliest possible physical time* at which reaction n can fire at the k -th tag. This quantity captures readiness, not necessarily whether a reaction actually fires; that is captured by the execution time function described below.

We introduce the following additional variables:

- $\mathcal{L}_{ji}(k)$: the *apparent latency* from reaction j to reaction i at the k -th tag.
- $e_n(k)$: the worst-case execution time (WCET) of reaction n at the k -th tag. If reaction n does not fire at the k -th tag, then $e_n(k) = 0$; otherwise $e_n(k) > 0$. This convention allows the execution time function to encode whether a reaction actually fires at a given tag.
- $t(k)$: the logical time value of the k -th tag.

Encoding System Constraints

We now derive when each reaction can fire at the k' -th tag. Consider reaction A1:

$$x_{A1}(k') = \max(x_{A1}(k) + e_{A1}(k), x_{A2}(k) + e_{A2}(k), t(k')). \quad (2.9)$$

The three terms correspond to the following constraints:

1. *Barrier synchronization*: $A1$ at the k' -th tag must wait for itself at the prior k -th tag to finish executing.
2. *Barrier synchronization*: $A1$ must also wait for $A2$ at the prior k -th tag to finish, since they reside on the same reactor.
3. *Logical time*: $A1$ must wait for physical time to reach the k' -th tag's logical time.

For reaction $A2$, the constraints arise from priorities and zero-delay connections:

$$x_{A2}(k') = \max(x_{A1}(k') + e_{A1}(k'), x_{B1}(k') + e_{B1}(k')). \quad (2.10)$$

Due to priority ordering, $A2$ must wait for $A1$ to finish at the same k' -th tag. Due to the zero-delay connection from B to A , $A2$ must also wait for $B1$ to finish at the same tag.

The constraints for reactor B are symmetric:

$$x_{B1}(k') = \max(x_{B1}(k) + e_{B1}(k), x_{B2}(k) + e_{B2}(k), t(k')), \quad (2.11)$$

$$x_{B2}(k') = \max(x_{A1}(k') + e_{A1}(k'), x_{B1}(k') + e_{B1}(k')). \quad (2.12)$$

Max-Plus Matrix Formulation

To cast the system of equations into a linear form amenable to analysis, we use the max-plus algebra, where $\oplus := \max$ and $\otimes := +$, with $\epsilon := -\infty$ as the zero element.

Since $x_{A1}(k')$ and $x_{B1}(k')$ depend only on quantities from the k -th tag (and on $t(k')$), they are already in a suitable form. For $x_{A2}(k')$ and $x_{B2}(k')$, we need to eliminate the dependence on $x_{A1}(k')$ and $x_{B1}(k')$. We walk through the algebraic steps for $x_{A2}(k')$ in detail; $x_{B2}(k')$ follows by symmetry.

Step 1: Substitute. Replace $x_{A1}(k')$ and $x_{B1}(k')$ in Equation (2.10) with their definitions from Equations (2.9) and (2.11):

$$x_{A2}(k') = \max \left(\underbrace{\max(x_{A1}(k) + e_{A1}(k), x_{A2}(k) + e_{A2}(k), t(k'))}_{x_{A1}(k')}, e_{A1}(k'), \right. \\ \left. \underbrace{\max(x_{B1}(k) + e_{B1}(k), x_{B2}(k) + e_{B2}(k), t(k'))}_{x_{B1}(k')}, e_{B1}(k') \right).$$

Step 2: Distribute. Distribute $e_{A1}(k')$ and $e_{B1}(k')$ into the inner max operators, using the identity $\max(a, b) + c = \max(a + c, b + c)$:

$$x_{A2}(k') = \max \left(\max(x_{A1}(k) + e_{A1}(k) + e_{A1}(k'), x_{A2}(k) + e_{A2}(k) + e_{A1}(k'), \right. \\ \left. t(k') + e_{A1}(k')), \right. \\ \left. \max(x_{B1}(k) + e_{B1}(k) + e_{B1}(k'), x_{B2}(k) + e_{B2}(k) + e_{B1}(k'), \right. \\ \left. t(k') + e_{B1}(k')) \right).$$

Step 3: Flatten. Since $\max(\max(\dots), \max(\dots)) = \max(\dots, \dots)$, we remove the inner max operators:

$$x_{A2}(k') = \max \left(x_{A1}(k) + e_{A1}(k) + e_{A1}(k'), x_{A2}(k) + e_{A2}(k) + e_{A1}(k'), \right. \\ \left. t(k') + e_{A1}(k'), x_{B1}(k) + e_{B1}(k) + e_{B1}(k'), \right. \\ \left. x_{B2}(k) + e_{B2}(k) + e_{B1}(k'), t(k') + e_{B1}(k') \right).$$

Step 4: Apply symmetry. Performing the same substitution for $x_{B2}(k')$ using Equations (2.9) and (2.11) yields an identical expression, since $A2$ and $B2$ share the same dependency structure through the zero-delay connections. Collecting all four equations and separating terms that depend on $X(k)$ from terms that depend on $t(k')$, we arrive at the max-plus linear difference equation

$$X(k') = M(k, k') \otimes X(k) \oplus \Gamma^*(k') \otimes \bar{t}(k'), \quad (2.13)$$

where $k' = k + 1$ and $\bar{t}(k')$ is the vector whose entries are all $t(k')$. Writing out the matrices explicitly, with row and column labels indicating the source and destination reactions (entry (i, j) records how reaction j constrains reaction i):

$$M(k, k') = \begin{array}{c} \begin{array}{cccc} & A1 & A2 & B1 & B2 \end{array} \\ \begin{array}{l} A1 \\ A2 \\ B1 \\ B2 \end{array} \left[\begin{array}{cccc} e_{A1}(k) & e_{A2}(k) & \epsilon & \epsilon \\ e_{A1}(k) \otimes e_{A1}(k') & e_{A2}(k) \otimes e_{A1}(k') & e_{B1}(k) \otimes e_{B1}(k') & e_{B2}(k) \otimes e_{B1}(k') \\ \epsilon & \epsilon & e_{B1}(k) & e_{B2}(k) \\ e_{A1}(k) \otimes e_{A1}(k') & e_{A2}(k) \otimes e_{A1}(k') & e_{B1}(k) \otimes e_{B1}(k') & e_{B2}(k) \otimes e_{B1}(k') \end{array} \right] \end{array}$$

and

$$\Gamma^*(k') = \begin{array}{c} \begin{array}{cccc} & A1 & A2 & B1 & B2 \end{array} \\ \begin{array}{l} A1 \\ A2 \\ B1 \\ B2 \end{array} \left[\begin{array}{cccc} 0 & \epsilon & \epsilon & \epsilon \\ e_{A1}(k') & 0 & e_{B1}(k') & \epsilon \\ \epsilon & \epsilon & 0 & \epsilon \\ e_{A1}(k') & \epsilon & e_{B1}(k') & 0 \end{array} \right]. \end{array}$$

The first term $M(k, k') \otimes X(k)$ captures the effects of reaction execution times from the prior tag, while the second term $\Gamma^*(k') \otimes \bar{t}(k')$ captures whether logical time lags behind physical time.

Factoring M and Recovering Proposition 2.1

The matrix $M(k, k')$ can be factored as $M(k, k') = \Gamma^*(k') \otimes B(k)$, where $\Gamma^*(k')$ is the intra-tag dependency closure defined above and

$$B(k) = \begin{array}{c} \begin{array}{cccc} & A1 & A2 & B1 & B2 \end{array} \\ \begin{array}{l} A1 \\ A2 \\ B1 \\ B2 \end{array} \left[\begin{array}{cccc} e_{A1}(k) & e_{A2}(k) & \epsilon & \epsilon \\ e_{A1}(k) & e_{A2}(k) & \epsilon & \epsilon \\ \epsilon & \epsilon & e_{B1}(k) & e_{B2}(k) \\ \epsilon & \epsilon & e_{B1}(k) & e_{B2}(k) \end{array} \right] \end{array}$$

is the *barrier synchronization matrix*, capturing the constraint that all reactions on a reactor *federate* must complete before the federate advances to the next tag. Observe that federated execution, under which a federate advances its tag independent of others, gives $B(k)$ the block diagonal structure described in Section 2.3: ordering the reactions as $(A1, A2, B1, B2)$ groups the two reactions of reactor federate A in the first two rows/columns and the two reactions of reactor federate B in the last two. The top-left 2×2 block captures reactor A 's local barrier, the bottom-right 2×2 block captures reactor B 's local barrier, and the two off-diagonal 2×2 blocks are filled with ϵ , reflecting that A and B advance their tags independently. This block-diagonal form is specific to decentralized execution, where each reactor enforces only its own local barrier. If the program were executed in a non-federated single process, then every reaction at tag k' would depend on every reaction at tag k , and $B(k)$ would instead take the form

$$B_{\text{global}}(k) = \begin{array}{c} \begin{array}{cc} & \begin{array}{cc} A1 & A2 & B1 & B2 \end{array} \\ \begin{array}{c} A1 \\ A2 \\ B1 \\ B2 \end{array} & \begin{bmatrix} e_{A1}(k) & e_{A2}(k) & e_{B1}(k) & e_{B2}(k) \\ e_{A1}(k) & e_{A2}(k) & e_{B1}(k) & e_{B2}(k) \\ e_{A1}(k) & e_{A2}(k) & e_{B1}(k) & e_{B2}(k) \\ e_{A1}(k) & e_{A2}(k) & e_{B1}(k) & e_{B2}(k) \end{bmatrix} \end{array} \end{array},$$

where every column is fully populated with the same execution time. The difference between the two forms captures, concretely, the price of running everything in a single process: the off-diagonal entries that are ϵ under decentralized execution become non- ϵ barriers under global barrier synchronization, tying every reaction's progress to the slowest reaction in the entire system.

This factorization aligns with the structure in [112]: Γ^* encodes the same-tag propagation of constraints through the reaction graph, while B encodes cross-tag barrier synchronization.

Substituting $M(k, k') = \Gamma^*(k') \otimes B(k)$ into Equation (2.13) and applying the distributivity of $\otimes$ over $\oplus$, we obtain Proposition 2.1:

$$\begin{aligned} X(k') &= \Gamma^*(k') \otimes B(k) \otimes X(k) \oplus \Gamma^*(k') \otimes \bar{t}(k') \\ &= \Gamma^*(k') \otimes \left(B(k) \otimes X(k) \oplus \bar{t}(k') \right). \end{aligned}$$

2.6 Case Study: Deriving Coordination Policy

We now apply Proposition 2.1 to a common pattern in cyber-physical systems: remote sensor polling. Consider the system shown in Figure 2.8, consisting of a **Controller** reactor and a **Sensor** reactor deployed as separate federates. The controller has a timer with offset 0 and period $p = 1$ s that triggers a reaction to send a polling command to the sensor. The sensor processes the command and returns data to the controller through a connection with logical delay $d = 1$ s. Since the delay equals the timer period, the sensor's response from tag k arrives at the controller precisely at tag k' , coinciding with the next timer tick.

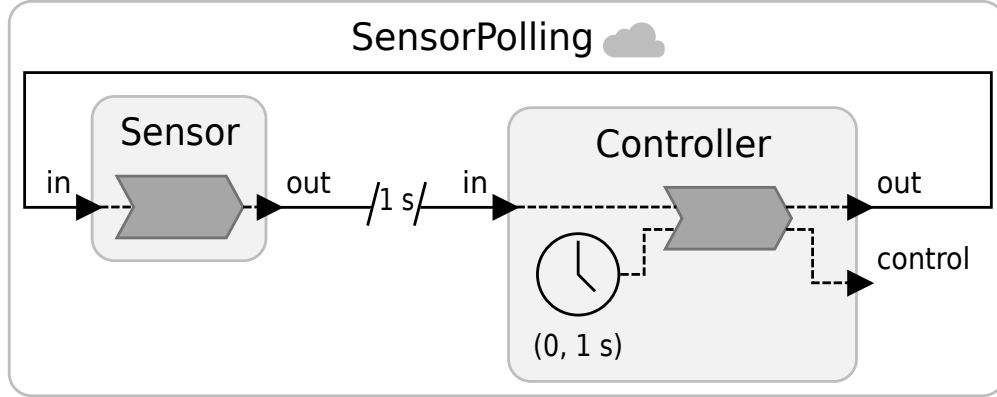


Figure 2.8: A sensor polling system: the **Controller** sends polling commands to the **Sensor** via a zero-delay connection and receives responses through a connection with a 1 s logical delay.

Under normal conditions, the sensor responds quickly. However, the sensor occasionally takes a long time to return data (*e.g.*, due to nondeterministic network latencies or the time needed to build a LiDAR point cloud). When deployed as a federated program using decentralized coordination, the controller must decide how long to wait for the sensor’s delayed response before safely proceeding to the next tag. The question is: *How should the maxwait parameters be set to ensure determinism?*

Modeling the System

We model the system using a state vector

$$X(k) = \begin{bmatrix} x_C(k) \\ x_S(k) \end{bmatrix},$$

where $x_C(k)$ and $x_S(k)$ represent the earliest physical start times of the controller’s reaction and the sensor’s reaction at the k -th tag, respectively. As before, $e_C(k)$ and $e_S(k)$ denote the reaction execution times, and $t(k)$ denotes the logical time of the k -th tag.

Since the logical delay d equals the timer period p , both the timer event and the delayed sensor response arrive at the same tag. At every tag after the first, both reactions fire: the controller processes the sensor’s response from the previous tag while simultaneously issuing a new polling command, and the sensor begins processing the new command.

Encoding System Constraints

Consider when each reaction can fire at the k' -th tag. For the controller:

$$x_C(k') = \max(x_C(k) + e_C(k), x_S(k) + e_S(k), t(k')). \quad (2.14)$$

The three terms correspond to the following constraints:

1. *Barrier synchronization*: the controller must finish its own reaction at the k -th tag.
2. *Cross-tag data dependency*: the controller at the k' -th tag processes the sensor's response from tag k , so the sensor must have finished at tag k . This constraint arises because the delayed connection ($d = p$) creates a dependency from the sensor's completion at tag k to the controller's readiness at tag k' .
3. *Logical time*: the controller must wait for physical time to reach $t(k')$.

For the sensor:

$$x_S(k') = \max(x_C(k') + e_C(k'), x_S(k) + e_S(k)). \quad (2.15)$$

The first term is a same-tag dependency: the sensor must wait for the controller to finish sending the polling command at the k' -th tag (the zero-delay connection from controller to sensor). The second term is barrier synchronization: the sensor must finish its own reaction at the k -th tag.

Deriving the Max-Plus Linear Form

We eliminate the dependence of $x_S(k')$ on $x_C(k')$ by substituting Equation (2.14) into Equation (2.15). First, note that $x_C(k') \geq x_S(k) + e_S(k)$, so $x_C(k') + e_C(k') \geq x_S(k) + e_S(k) + e_C(k') \geq x_S(k) + e_S(k)$. The self-barrier term for the sensor is therefore dominated, and Equation (2.15) simplifies to

$$x_S(k') = x_C(k') + e_C(k'). \quad (2.16)$$

Substituting Equation (2.14) and distributing:

$$\begin{aligned} x_S(k') &= \max(x_C(k) + e_C(k), x_S(k) + e_S(k), t(k')) + e_C(k') \\ &= \max(x_C(k) + e_C(k) + e_C(k'), x_S(k) + e_S(k) + e_C(k'), t(k') + e_C(k')). \end{aligned}$$

Using the max-plus operators $\oplus := \max$ and $\otimes := +$, we write the matrix form directly in the factored shape of Proposition 2.1:

$$X(k') = \Gamma^*(k') \otimes \left(B(k) \otimes X(k) \oplus \bar{t}(k') \right), \quad (2.17)$$

where

$$\Gamma^*(k') = \begin{array}{c} C \quad S \\ \begin{bmatrix} 0 & \epsilon \\ e_C(k') & 0 \end{bmatrix} \end{array}, \quad B(k) = \begin{array}{c} C \quad S \\ \begin{bmatrix} e_C(k) & e_S(k) \\ e_C(k) & e_S(k) \end{bmatrix} \end{array}, \quad \bar{t}(k') = \begin{array}{c} C \\ \begin{bmatrix} t(k') \\ t(k') \end{bmatrix} \end{array}.$$

Here Γ^* captures the same-tag dependency (the sensor depends on the controller at the same tag), and B captures cross-tag constraints: each row contains both $e_C(k)$ (self-barrier of the

controller) and $e_S(k)$ (the data dependency through the delayed connection from the sensor). Note that B has identical rows, not because both reactions reside on the same reactor, but because the sensor's constraints at tag k' are fully mediated through the controller at tag k' , which itself depends on both $e_C(k)$ and $e_S(k)$. Expanding Equation (2.17) component-wise recovers the scalar equations derived above.

Deriving Maxwaits from Trace Prediction

Before proceeding, recall the element-wise *subtraction* operator $\oslash$ (i.e., $a \oslash b = a - b$), which we will use to express the lag between physical firing times and logical timestamps.

Deriving maxwaits. The unavailability $\bar{A}(k')$ can be interpreted as the *exact processing offset* $O(k')$ required to maintain determinism at the k' -th tag:

$$O(k') = \bar{A}(k') = \Gamma^*(k') \otimes \left(B(k) \otimes X(k) \oslash \bar{t}(k') \oplus \vec{0} \right). \quad (2.18)$$

This formulation corresponds to the processing offset equation $O = Z \oplus \Gamma O = \Gamma^* Z$ from [112], where $\vec{0}$ reflects the requirement to track real time. Proposition 2.1 is more precise: it includes not only the real-time attachment ($\vec{0}$), but also the dynamics from the prior tag ($B(k) \otimes X(k)$). If $B(k) \otimes X(k) > \bar{t}(k')$, the system cannot proceed at full speed even with real-time tracking.

To obtain a single, static processing offset O suitable as an execution policy, corresponding to the *maxwait* values used in LINGUA FRANCA's decentralized coordination, we solve

$$O = \max_{k \in \mathbb{N}} O(k).$$

Using O as an execution policy changes the trace itself. Without an execution policy, when an upstream reaction n_j does not produce outputs, the downstream reaction n_i would have $e_i(k) = 0$. However, under decentralized coordination, the downstream reaction must wait for the processing offset, effectively setting $e_i(k) = O$.

Concrete example. Suppose the controller executes in $e_C = 1$ ms and the sensor normally executes in $e_S = 101$ ms. In steady state with no lag, $x_C(k) = t(k)$ and $x_S(k) = t(k) + e_C$. Now suppose at tag k the sensor experiences a fault and takes $e_S(k) = 1120$ ms to respond. Applying Equation (2.14) at the next tag:

$$\begin{aligned} x_C(k') &= \max(t(k) + 1, t(k) + 1 + 1120, t(k) + 1000) \\ &= t(k) + 1121. \end{aligned}$$

The controller's lag at tag k' is therefore $x_C(k') - t(k') = t(k) + 1121 - (t(k) + 1000) = 121$ ms. This means the maxwait for the controller must be at least 121 ms to ensure that the

controller waits long enough for the sensor’s delayed response before safely advancing, thereby preserving determinism even when the sensor occasionally exceeds its normal execution time.

Figure 2.9 shows the execution trace predicted by Proposition 2.1 when the maxwait is applied. The trace confirms that, when stale data occurs, the controller’s processing offset is sufficient to wait for the sensor’s delayed response, maintaining determinism throughout the execution.

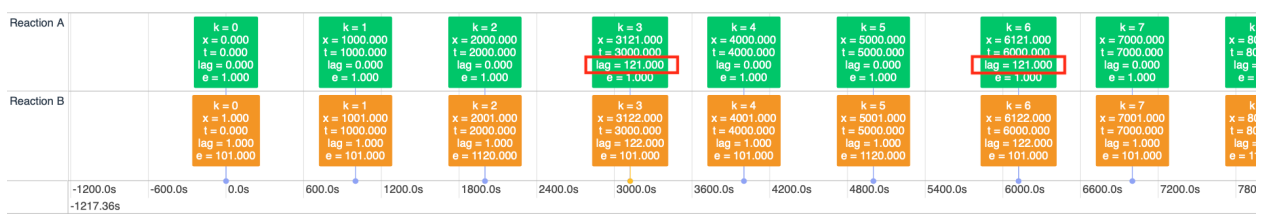


Figure 2.9: Execution trace predicted by Proposition 2.1 for the sensor polling system. When the sensor occasionally exceeds its normal execution time, the controller’s maxwait of 121 ms ensures it waits long enough for the delayed response, preserving determinism.

2.7 Conclusion

This chapter has developed a framework for system-level timing analysis, building on the original CAL theorem [112, 109] and tailored to the reactor model of computation and to the LINGUA FRANCA coordination language [129, 123]. Whereas the original CAL theorem establishes worst-case algebraic bounds on the trade-off between consistency, availability, and apparent latency across an entire execution, Proposition 2.1, at the core of this chapter, operates at the granularity of individual reactions and models their physical firing time instants *exactly*. By expressing each reaction’s firing time as a max-plus linear function of input tags, timestamps, measured execution times, and measured network latencies, the framework collapses the pessimism inherent in worst-case bounds and lifts the CAL relation from a system-wide inequality to a per-reaction equality along a concrete trace.

The technical core of the chapter is the construction of a structured max-plus system matrix M whose entries encode the causal, dataflow, and timing constraints of a federated LF program, and whose factorization exposes Proposition 2.1 at every reaction. This form reveals, for each reaction, precisely which incoming dependency is the binding constraint on its firing time, and how much consistency one would have to relinquish on that dependency to recover availability. The factorization thereby turns the consistency-availability trade-off from a global design decision into a local, reaction-by-reaction calculation that a designer or a compiler can reason about directly. Through the case studies in Sections 2.4 and 2.6 we have shown how this machinery supports two distinct engineering uses of the framework: first, the eigenvalue of M characterizes the *critical path* of a distributed reactor program,

identifying the dominant feedback loop in the communication topology whose cycle weight governs steady-state throughput and whose edges are the right targets for latency reduction or inconsistency relaxation; and second, Proposition 2.1 enables the systematic derivation of *maxwait* parameters (safe-to-advance and safe-to-assume-absent offsets) for decentralized coordination directly from trace data, replacing guesswork and conservative hand-tuning with principled, measurement-driven values.

Taken together, these contributions advance the CAL framework from abstract trade-off analysis into a constructive, trace-aware design calculus for distributed cyber-physical systems. The shift from worst-case reasoning to exact trace modeling brings several practical benefits. It allows past execution data collected from tracing infrastructure to be analyzed retrospectively, which the original CAL theorem could not usefully do. It enables more accurate predictions of future behavior under realistic, rather than adversarial, assumptions about execution times and network delays. It also closes the loop between the high-level specification of availability and consistency requirements expressed in LINGUA FRANCA’s **deadline** and **after** constructs, and the low-level runtime parameters that the federated coordinator must set in order to realize those requirements on a given deployment. In this sense, the framework provides the missing design-time feedback that the introduction of this chapter identified as a central difficulty in cyber-physical system engineering: given a program, a target platform, and measured or estimated timing characteristics, the designer can now compute, not merely bound, the consistency-availability operating point of the system.

Several avenues remain open for future work. First, the max-plus eigenvalue analysis presented here assumes a well-defined system matrix $M(k', k)$, with concrete timing constraints for all indices k and k' . However, when designing a system, engineers often work with probabilistic models. Can the framework be extended into supporting a probabilistic model for M ? Second, although this chapter has focused on deriving maxwaits and identifying critical paths, the same algebraic framework is a natural substrate for *synthesis*: given availability targets and a latency budget, one could automatically synthesize the minimal **after** delays, task-to-core mappings, or physical network topologies required to make the program schedulable, or automatically place reactors across end devices, edge computers, and the cloud to minimize a weighted combination of unavailability and inconsistency. Third, integrating the framework with runtime monitoring and adaptive coordination would turn the framework into a live control loop: as measured latencies drift, the maxwaits and inconsistency allowances could be updated on the fly to preserve determinism and meet deadlines. Pursuing these directions would further cement the framework’s role as both an analytical lens and a constructive tool for the design, deployment, and operation of correct distributed real-time cyber-physical systems.

Chapter 3

Formal Verification of Engineering Models

“The map is not the territory.”

—*Alfred Korzybski, Science and Sanity, 1933.*

3.1 Engineering Models and Scientific Models

In science, the objective of a model is to match a real-world system. In contrast, in engineering, the objective of a real-world system (an implementation) is to match a model [114]. A digital logic diagram, for example, is an engineering model; a circuit gets constructed to match the model. Differential equations giving currents and voltages in a circuit over time are commonly used as a scientific model; the equations are crafted to match the physical circuit. Even these equations could be used as an engineering model, giving, for example, a specification of an analog circuit. The actual components used will not be perfectly linear and will not have resistance, capacitance, and inductance that perfectly match the model. When using models, it is important to understand whether they are being used as engineering models or scientific models. Are the resistances wrong in the model or in the physical circuit? One of the two must be true.

This chapter argues that, because formal verification necessarily reasons about models, it is important to distinguish which parts of a formal model are engineering models (*i.e.*, specifications) and which are scientific models (*i.e.*, approximations of the systems or environments being modeled). These two distinct styles of modeling should not be conflated, and combining them requires a principled approach. George Box famously said, “all models are wrong, but some are useful” [23]. Here, Box was referring to scientific models. The corresponding statement for engineering models is, “all physical, real-world implementations are wrong, but some are useful” [107]. An engineering model is, or at least attempts to

be, right, not wrong. It is a specification, an idealization, and an implementation can only approximate it.

Formal verification, by its nature of being formal, can only make assertions about *models*. For scientific models, therefore, verification results should be understood as conditional on the adequacy of the model as an approximation of the system or environment being modeled. On the other hand, when a model serves as an engineering specification, verification can establish properties of the design itself, which can be extremely useful. Indeed, many success stories for formal verification concern synchronous digital logic and software, where the verified artifact is naturally understood as an engineering model. The physical realization in both cases is “electrons sloshing in silicon” [114], about which nothing is proved.

In this chapter, we examine TIMED REBECA (TR), a language and formalism for describing and analyzing timed systems. *We argue that the timing features of TR are more useful when viewed as an engineering model than when viewed as a scientific model.* To support the case, we show the relationship between the timing constructs of TR and an implementation (vs. modeling) language with similar timing constructs, LINGUA FRANCA (LF). We show that, with the addition of priorities for message handlers, TR can model the timing constructs of LINGUA FRANCA and therefore can be used to effectively formally verify engineering models specified using LF.

For example, the TR **after** delay gives a time offset between the sending and the receiving of a message between actors. If this **after** delay is intended to model physical communication latencies, then it is being used as a scientific model. The model is correct only if the physical communication latencies actually match. If, on the other hand, it is used to specify a logical offset between the sender’s view and the receiver’s view of changing data, then it is being used as an engineering model. The implementation is correct only if it actually delivers the specified logical offset. Such delays are used in LF in the latter sense, not the former, whereas in TR, we have seen them used both ways.

A significant limitation remains, however. Formal verification can only be performed on closed models. For systems that interact with an uncontrolled environment that is not designed, we have no choice but to provide a scientific model of that environment. Any proof of the correctness of a design, therefore, is predicated on the assumptions about the environment. These assumptions are made evident by making a clear separation between models of the environment and models of the system under design. The weaker those assumptions, the stronger the confidence provided by the verification results.

3.2 Scientific and Engineering Models in TIMED REBECA

TIMED REBECA [157] is an extension of the REBECA modeling language [173] (an implementation of the Hewitt-Agha actor model [71, 3, 4]) that adds a notion of time. In this extension, time is an integer, messages are sent and arrive at (possibly distinct) times and

```

1 env int requestDeadline, checkIssuedPeriod,
  retryRequestPeriod, newRequestPeriod,
  serviceTime1, serviceTime2;
2
3 reactiveclass Agent {
4   knownrebecs { TicketService ts1; TicketService
    ts2; }
5   statevars { int attemptCount;
6     boolean ticketIssued; int token; }
7   msgsrv initial() {
8     self.findTicket(ts1); }
9   msgsrv findTicket(TicketService ts){
10    attemptCount += 1; token += 1;
11    ts.requestTicket(token) deadline(
12    requestDeadline);
13    // Check if a reply to the
14    // request has been received.
15    self.checkTicket() after(checkIssuedPeriod);}
16  msgsrv checkTicket() {
17    if (!ticketIssued
18    && attemptCount == 1) {
19      self.findTicket(ts2);
20    } else if (!ticketIssued
21    && attemptCount == 2) {
22      self.retry() after(retryRequestPeriod);
23    } else if (ticketIssued) {
24      ticketIssued = false;
25      self.retry() after(newRequestPeriod);
26    }
27  msgsrv ticketIssued(int tok){
28    if (token == tok) {
29      ticketIssued = true; }}
30  msgsrv retry() {
31    attemptCount = 0;
32    self.findTicket(ts1);}
33 }
34 reactiveclass TicketService {
35   knownrebecs { Agent a; }
36   msgsrv initial() { }
37   msgsrv requestTicket(int token){
38     // The ticket service sends
39     // the reply after a non-
40     // deterministic delay of
41     // either serviceTime1 or
42     // serviceTime2.
43     int wait = ?(serviceTime1,serviceTime2);
44     delay(wait);
45     a.ticketIssued(token);
46   }
47 }
48 main {
49   Agent a(ts1, ts2):();
50   TicketService ts1(a):();
51   TicketService ts2(a):();
52 }

```

Figure 3.1: TIMED REBECA model for a ticket service, from [157].

message handlers (optionally) take time to execute. Three operators are added to the REBECA language to model time: **delay**, **deadline**, and **after**. A delay statement models the passing of time of an actor during the execution of a message server. The **after** operator indicates time that elapses before a message is delivered to its destination. The **deadline** operator models message loss if the destination is busy due to **delay** statements and does not handle an incoming message before the deadline expires.

In addition to these operators, TIMED REBECA strengthens the semantics that define the order in which message handlers are invoked. Because of these operators, each message delivered to an actor has a time associated with it, and the destination actor will handle messages in temporal order, unless it is busy due to a **delay** statement, in which case it will handle the message at the time the delay expires. If two messages are delivered with the same time, then the handler order is nondeterministic unless the programmer specifies priorities. The ability to specify priorities was added later specifically to support the kind of modeling we describe in this chapter [175, 84].

Reynisson et al. [157] give a ticket service example (Fig. 3.1), where an agent accepts requests for tickets from an environment and issues requests for a ticket to two ticket services. If the first ticket service does not respond with a ticket within a specified amount

of time (defined by the **checkIssuedPeriod** environment variable, defined on line 1 and used on line 14), then it tries the second ticket service. The **TicketService** actor uses the **delay** operator to nondeterministically delay responding by either **serviceTime1** or **serviceTime2** (lines 43 and 44). Here, there is a mix of scientific and engineering modeling. The value of **checkIssuedPeriod** is an engineering *specification*. It defines how long the **Agent** actor should wait before giving up on the requested ticket service. On the other hand, the **serviceTime1** or **serviceTime2** variables model communication and computation time, which are *assumptions* derived from the environment through scientific means (measurement, for example). When performing verification on this model, the results of the verification can only be considered valid for these two particular assumed delays. If the actual observed delays deviate from these assumed delays, which could always happen due to the nature of scientific modeling, the verification model is no longer applicable.

Later, Sirjani and Khamespanah note the distinction between engineering models and scientific models and describe a TIMED REBECA example for components of a network-on-chip [174]. Specifically, the model consists of a Manager, which generates network traffic, and a Router, which routes traffic according to a specified policy. In this example, the precise times given in **after**, **delay**, and **deadline** statements can be interpreted as specifications for the time taken by a synchronous digital circuit implementation of the router. No time units are given in the paper, but in this engineering-model interpretation, they could be clock cycles or SI units (such as nanoseconds) as measured by the clock of a synchronous digital circuit.

We argue here that the TIMED REBECA constructs, **delay**, **deadline**, and **after**, are better used as part of an engineering model, where timing constraints are specifications of desired behavior. They can also be used in scientific models, for example when modeling an uncontrolled environment, but single timing values are a poor fit for phenomena such as network delays, computation times, and clock synchronization error, which are better represented using ranges or other forms of uncertainty. Using **after** to model communication delays, for example, is questionable for systems using modern network communication. The delays can be nondeterministically chosen, but for verification to be tractable, a small number of possible delays must be used. Moreover, these delays refer to clock measurements of time at distinct points in the network. Reynisson et al. [157] assume perfect (or near perfect) clock synchronization, but perfect clock synchronization is not possible. On the other hand, when these numbers refer to a *logical* clock instead of physical clocks, and the time ordering of events is a *semantic* property of the system rather than an emergent property of the implementation, then verification results can be trusted subject only to the condition that the system implementation correctly realizes the semantics. We show how to do that in this chapter.

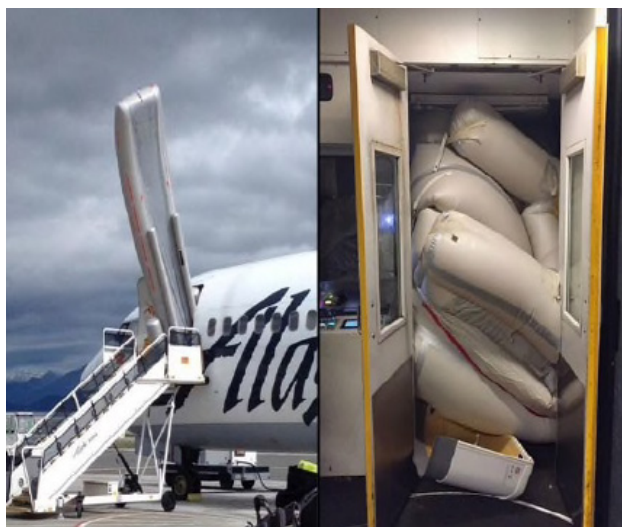


Figure 3.2: Two accidental deployments of emergency escape slides (image from *The Telegraph*, Sept. 9, 2015).

3.3 Running Example

In this section, we introduce an example (somewhat artificial and grossly oversimplified) which is a distributed system in which the order of events is of critical importance. There are many practical examples that have this character, but we pick this one because it is vivid, simple, and challenging to get right, all at the same time.

Consider automating the opening and closing of an aircraft door. Modern passenger planes are equipped with emergency escape slides that are automatically deployed if the door is opened while the door is “armed.” When the doors are closed for departure, flight attendants arm them. At arrival, once they have verified that a ramp is right outside the door, the flight attendants disarm the door before an operator opens it. The process currently is manual, with a red or green flag displayed in a window on the door to indicate whether it is armed or not.

Suppose we wish to automate the process so that the door can be opened remotely. There might be multiple systems on the aircraft that issue a command to open the door. Some of these, such as a cockpit control, might want to ensure that a ramp is present and the door is disarmed before it is opened, while others, such as a fire alarm system, might want to open the door whether it is armed or not.

The door will be equipped with a networked microprocessor that provides just two services, **disarm** and **open** (we leave out arming and closing, which are not needed to make our points). Assume the door starts in the armed and closed state. What should the microprocessor do when it receives an **open** command over the network? It is critical to be sure that if the **open** command has an associated **disarm** command, or if there is a ramp present at

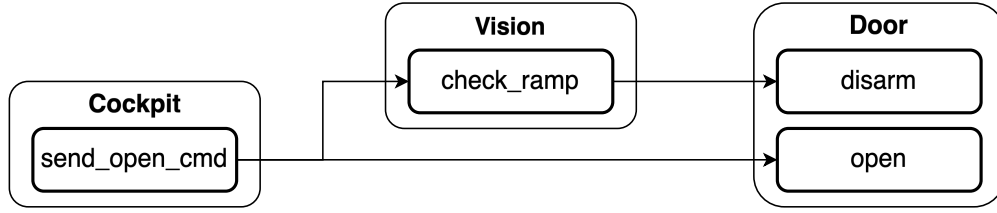


Figure 3.3: Aircraft door system schematic using the Actor model.

the door, that the door should be disarmed before it is opened. We have to guard against errors due to networking or computation delays, otherwise we might find ourselves in the life-threatening situations depicted in Fig. 3.2.

Suppose that when a cockpit control issues an instruction to open the door, the actual opening is delayed until a ramp is present. As shown in Fig. 3.3, a computer vision system might check for a ramp. Suppose further that the Cockpit, Vision, and Door components are realized as Hewitt-Agha actors. We immediately have a problem because the order of message delivery and handling in the Hewitt-Agha model is nondeterministic. It is highly sensitive to delays in the network and in intermediate computations. With the design in Fig. 3.3, the **open** message will very likely arrive before the **disarm** message, resulting in catastrophe.

Fig. 3.4 shows a TIMED REBECA model that uses a combination of temporal semantics and explicit priorities to ensure correct ordering of message handlers. The constructor for the Cockpit component starts things off by sending itself a **send_open_cmd** on line 4. It then repeatedly invokes this same message handler every 100 time units on line 11. This, effectively, models an environment, where the time units for the repeat may represent a worst-case maximum frequency for events. We have, therefore, a scientific model of an environment that encodes an assumption about its worst-case behavior. To trust any verification results with this model, we must convince ourselves that any time intervals larger than this will not result in violations of the properties of interest.

In this (simplified) model, the Door actor has a state variable **status** (line 28) that takes on the value 3 if the door opens and the slides are deployed (Fig. 3.2). Since this model only includes the cockpit control and assumes that the Vision actor always sees a ramp (line 19), we would like to verify that **status** never reaches value 3. Using Afra, the TIMED REBECA model-checking tool, we can verify that this requirement is satisfied. However, the requirement would not be satisfied without the **@priority** annotations on lines 31, 36, 46, 47, and 48. The priority annotations are a recent addition in TIMED REBECA [175, 84]. They can be specified at the level of message servers (lines 31, 36) and at the level of actor instances (lines 46, 47, and 48). Semantically, a smaller priority value indicates higher execution priority. Components with the same priority get executed in a nondeterministic order when they are enabled. Priorities specified on message servers are treated separately

```

1 reactiveclass Cockpit(10) {
2   knownrebecs { Vision v; Door d; }
3   Cockpit() {
4     self.send_open_cmd();
5   }
6   msgsrv send_open_cmd() {
7     // Send open message.
8     d.open(true);
9     v.check_ramp(true);
10    // Repeat after 100 ms.
11    self.send_open_cmd() after(100);
12  }
13 }
14 reactiveclass Vision(10) {
15   knownrebecs { Door d; }
16   msgsrv check_ramp(boolean trigger) {
17     // Use vision to detect ramps.
18     // Assume a ramp is present.
19     d.disarm(true);
20   }
21 }
22 reactiveclass Door(10) {
23   statevars {
24     // 0: closed_armed,
25     // 1: closed_disarmed,
26     // 2: open,
27     // 3: open_deployed
28     int status;
29   }
30   Door() { status = 0; }
31   @priority(1)
32   msgsrv disarm(boolean disarm) {
33     // Disarm the door.
34     if (status == 0) status = 1;
35   }
36   @priority(2)
37   msgsrv open(boolean open) {
38     // Actuate door opener.
39     if (status == 0)
40       status = 3;
41     else if (status == 1)
42       status = 2;
43   }
44 }
45 main {
46   @priority(1) Cockpit c(v, d):();
47   @priority(2) Vision v(d) :();
48   @priority(3) Door d() :();
49 }

```

Figure 3.4: TIMED REBECA model for the aircraft door system using priorities.

from priorities specified on actors.

In this model, the only temporal operator is on line 11. Hence, all message transport and message handling is assumed to be instantaneous. Without the priorities, there is a risk that the **open** message handler of Door will be invoked before the **check_ramp** handler of Vision or, if not, before the **disarm** handler of Door. This, again, would result in catastrophe.

In a distributed system, enforcing these priorities is not trivial. The LINGUA FRANCA language, described in the next section, infers priorities based on data dependencies and enforces them at runtime. Without LF, a distributed implementation of this design would be quite difficult to realize. Moreover, the enforcement aspect of the implementation would be error-prone, weakening the verification results. We may have formally verified the design, but without a strong assurance that the implementation matches the design, the verification result is not very useful.

Given the temporal semantics of TIMED REBECA, it is tempting to instead model the timing of an implementation and verify that model. It may be, for example, that instead of assuming unrealistic instantaneous computation and message transport, the delays experienced in a physical implementation will lead to a safe design. The TIMED REBECA model in Fig. 3.5 attempts to check this. To open the aircraft door, the **Cockpit** actor calls the **send_open_cmd** handler, which sends two messages. It first sends a **check_ramp** message to

```

1 reactiveclass Cockpit(10) {
2   knownrebecs {
3     Vision v; Door d;
4   }
5   Cockpit() {
6     self.send_open_cmd();
7   }
8   msgsrv send_open_cmd() {
9     // Send message to check ramp.
10    // Max network latency = 2ms.
11    v.check_ramp(true) after(2);
12    // Delay to avoid accidents.
13    delay(5);
14    // Send open message.
15    // Max network latency = 2ms.
16    d.open(true) after(2);
17    // Repeat after 100 ms.
18    self.send_open_cmd() after(100);
19  }
20 }
21 reactiveclass Vision(10) {
22   knownrebecs { Door d; }
23   msgsrv check_ramp(boolean trigger) {
24     // Computation time = 2 ms.
25     delay(2);
26     // Use vision to detect ramps.
27     // Assume a ramp is present.
28     // Max network latency = 2
29     d.disarm(true) after(2);
30   }
31 }
32 reactiveclass Door(10) {
33   statevars {
34     // 0: closed_armed,
35     // 1: closed_disarmed,
36     // 2: open,
37     // 3: open_deployed
38     int status;
39   }
40   Door() { status = 0; }
41   msgsrv disarm(boolean disarm) {
42     // Disarm the door.
43     if (status == 0) status = 1;
44   }
45   msgsrv open(boolean open) {
46     // Actuate door opener.
47     if (status == 0)
48       status = 3;
49     else if (status == 1)
50       status = 2;
51   }
52 }
53 main {
54   Cockpit c(v, d):();
55   Vision v(d) :();
56   Door d() :();
57 }

```

Figure 3.5: Using **after** and **delay** to avoid accidentally deploying emergency slides, model network latencies, and model execution time.

the Vision actor (line 11) using an **after** delay to model the network delay. It then waits 5 time units (line 13) before sending an **open** message (line 16), again using an **after** delay to model the network delay. The Vision actor models its computation time on line 25 and the network delay of its relay to Door on line 29. Again, we can use Afra to verify that the Door actor never reaches **status == 3**.

This model, however, exposes a critical flaw in this style of design. If we assume the **after** delays are worst-case scientific models of network delay, then we have a problem. If the **open** message from the Cockpit happens to experience less delay, say one time unit instead of two, then the model becomes nondeterministic, and the door may be opened before it is disarmed. To correct for this, we could increase the delay on line 13. In fact, this model mixes scientific models (lines 11, 16, 25, and 29) with engineering modeling (line 13). The scientific models must be interpreted as worst-case behaviors, and we have to do careful reasoning, separate from the formal verification, to assure ourselves that lower values will not hurt. On the other hand, the engineering model on line 13 has the opposite semantics. It must be interpreted as a lower bound, a *requirement* on the system design to ensure that

accidental deployment of emergency slides never occurs. The formal verification performed by Afra does not help with this reasoning but only tells us whether the desired condition is satisfied for the particular delays chosen.

The ultimate root cause of the problem is mixing *engineering* models with *scientific* models in an undisciplined way. We illustrate how to correct this by clearly separating the model of the environment from the model of the system under design, and by exclusively using engineering models for the system under design.

3.4 Overview of LINGUA FRANCA

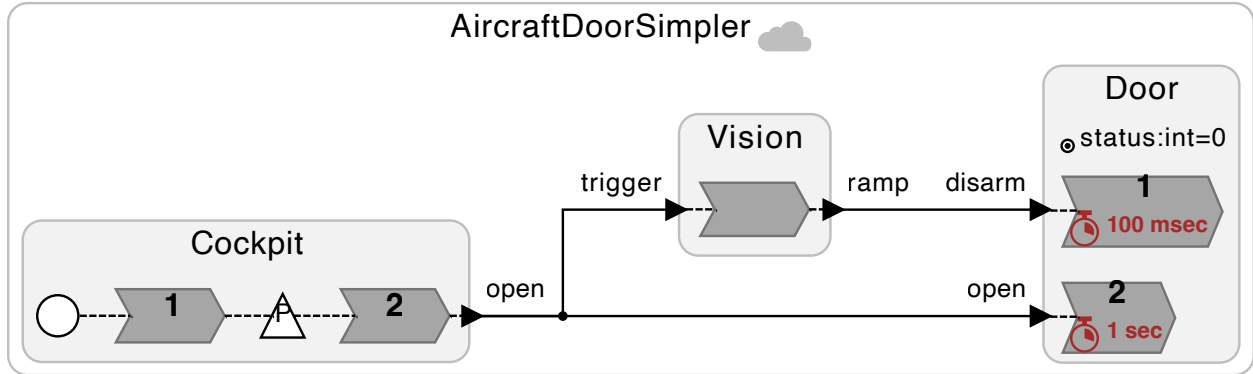
LINGUA FRANCA (LF) [129] is a polyglot coordination language designed to augment multiple mainstream programming languages (also called target languages), currently C, C++, Python, TypeScript, and Rust, with deterministic reactive concurrency and the capability to specify timed behavior. LF is supported by a runtime system that enables concurrent and distributed execution of reactive programs, which can be deployed on various platforms, including in the cloud, at the edge, in containers, and even on resource-constrained bare-metal embedded platforms.

A LINGUA FRANCA program defines interactions between components known as reactors [127], with the logic for each reactor written in plain target code. LINGUA FRANCA's code generator then produces one or more programs in the target language, which are compiled using standard toolchains. When the application has parallelism, it runs on multiple cores without losing determinism. For distributed applications, multiple programs and scripts are generated to deploy these programs on multiple machines and/or containers. The network communication fabric connecting these components is also synthesized as part of the code generation and compilation process.

Reactor-Oriented Programming

LINGUA FRANCA programs consist of reactors, which are stateful entities with event-driven routines. Reactors adopt advantageous semantic features from established models of computation, namely actors [5], logical execution time [86], synchronous reactive languages [18], and discrete event systems [110] (such as DEVS [194] and SystemC [118]). The reaction routines belonging to reactors can process inputs, generate outputs, alter the reactor's state, and schedule future events. Reactors resemble actors [5], which are software components that communicate through message passing. However, unlike traditional actors, these messages have timestamps, and the concurrent interaction of reactors is deterministic by default. Any nondeterministic behavior must be explicitly programmed if needed.

Figure 3.6 shows a LINGUA FRANCA program realizing the aircraft door service described in the previous section. The textual code shown at the bottom of the figure is LINGUA FRANCA code, and the diagram above it is automatically generated by the tools and updated dynamically as the code is edited. In this example, the main program is *feder-*



```

1 target C // or Python, Rust, etc.
2 federated reactor {
3   c = new Cockpit()
4   v = new Vision()
5   d = new Door()
6   c.open -> d.open
7   c.open -> v.trigger
8   v.ramp -> d.disarm
9 }
10 reactor Cockpit {
11   physical action command
12   output open: bool
13   reaction(startup) -> command {=
14     // Set up cockpit UI controls.
15     // Target language code here...
16   =}
17   reaction(command) -> open {=
18     // Target language code here.
19     // E.g., send open message.
20     lf_set(open, true);
21   =}
22 }
23 reactor Vision {
24   input trigger: bool
25   output ramp: bool
26   reaction(trigger) -> ramp {=
27     // Target language code here...
28   =}
29 }
30 reactor Door {
31   input open: bool
32   input disarm: bool
33   state status: int = 0
34   // This reaction should be first
35   reaction(disarm) {=
36     // Disarm the door.
37     if (self->status == 0) {
38       self->status = 1;
39     }
40   =} deadline(100 ms) {=
41     // Handle the deadline violation
42   =}
43   reaction(open) {=
44     // Actuate door opener.
45     if (self->status == 0) {
46       self->status = 3;
47     } else if (self->status == 1) {
48       self->status = 2;
49     }
50   =} deadline(1 s) {=
51     // Handle the deadline violation
52   =}
53 }

```

Figure 3.6: Aircraft door system schematic and LINGUA FRANCA specification.

ated (line 2), which means that each of the top-level reactors, **Cockpit**, **Vision**, and **Door**, will be generated as its own program in the target language (specified on line 1). These programs, called *federates*, can be containerized for better isolation and fault tolerance and/or distributed to distinct hardware, for example to exploit specialized hardware in the **Vision** reactor. This **Vision** reactor accesses a camera with a computer vision system that detects a ramp outside the door and disarms the door before opening it.

The grey chevrons inside reactors in the diagram represent *reactions*, which are triggered by inputs, timers, or events on an event queue and are able to produce outputs. The business logic inside reactions is written in the target language that is chosen. A typical implementation of these reactors could use, for example, the Python target, which can leverage existing packages such as Tensorflow lite for the vision component, if it is to be realized on the edge, or it could use an API to realize the vision component in the cloud, or it could use the C target together with OpenCV, the open computer vision library (this code would go on line 27). Legacy code and libraries are easy to use in LF by simply invoking their APIs in the reaction bodies.

In principle, a federated LF program could use different target languages for each of the federates, such as Python for the **Vision** federate and C or Rust for the other components, but this capability only exists currently in concept demonstration form.

Cockpit uses a common pattern for LF programs that react to asynchronous events from the environment. It has a *physical action*, defined on line 11, which is used to inject asynchronous external events from the environment. The reaction to the *startup* event on line 13 can be used to set up any external interactions, for example enabling an interrupt service routine to handle sensor events or cockpit switches. When these events occur, the service routine schedules the physical action, which gets assigned a timestamp drawn from a local physical clock. The code on line 17 will then be invoked in reaction to such events.

A key property of LINGUA FRANCA is that every reactor handles events in timestamp order, and when events bear the same timestamp, they will appear simultaneously at the reactor. Moreover, reactions are logically instantaneous. If they produce outputs in reaction to inputs, as the **Vision** does, those outputs have the same timestamp as the input. Consequently, LF semantics assures that the **Door** reactor will not react to an open command until the results of the **Vision** reactor are delivered to it. Moreover, when a reactor is presented with multiple simultaneous inputs, reactions will be invoked in the lexical order. Hence, given simultaneous **disarm** and **open** events, the **Door** reactor will react first to the **disarm** event (line 35) and then to the **open** event (line 43). As we have seen in the previous section, it is not trivial to ensure such behavior in an actor network, even one extended with time, as in TIMED REBECA. It is, nevertheless, possible, as we will show.

The **Door** reactor also includes two *deadline* declarations, which serve two purposes. First, they guide the LF scheduler to prioritize reaction invocations with nearer deadlines. Second, they provide fault handling code, on lines 40 and 50, which is invoked instead of the regular reaction code if and when the deadline is violated.

In this example, the federated reactors can be deployed on separate machines and reactions in different reactors can be executed by multiple threads in parallel. More importantly, for the same input values and timing, the partial order constraints on the execution of reactions ensure determinism. This modular design of reactors, determinism, and deployment flexibility make LF suitable for describing time-sensitive applications.

Specifying Timing Behavior

LINGUA FRANCA makes a distinction between two timelines, **logical time** and **physical time** [125]. **Logical time** is represented by a tag (timestamp and microstep in superdense time [35, 134]) that tracks the processing of events within the system. It is a marker for the sequence and timing of events as understood by the system’s logic. **Physical time** tracks the actual movement of physical clocks, not to be confused with the conceptual clocks used in synchronous-reactive models. In LF’s model of time, logical time, by default, lags behind physical time, meaning that the system’s logical processing waits for the physical time to advance before proceeding.

LINGUA FRANCA programs can explicitly specify a variety of timing behaviors in time-sensitive systems. These timing behaviors include time-triggered events, constraints on asynchronous events (such as minimum spacing), and deadlines. The timing behavior specification in LINGUA FRANCA allows deterministic execution for timed events and user inputs. Reactions to events are executed in order based on the logical time (i.e., the tag).

The *physical action* in the **Cockpit** reactor (line 11), represented in the diagram by a triangle with a “P”, captures external asynchronous inputs, assigning them a logical timestamp based on the physical time, as measured by a local clock. Lines 40 and 50 give deadlines, which specify a maximum acceptable gap between the logical time and physical time of a reaction invocation. These are also shown in the diagram as red markers in the reactions of the **Door** reactor.

Like TIMED REBECA, LINGUA FRANCA supports **after delays** on connections between reactors (not used in this example). These increment the timestamp of an event conveyed along the connection. They are part of the program logic, not a scientific model of communication delays. This manipulation can be used to ameliorate the effects of the fundamental trade-off between consistency and availability in a distributed system [108].

For federated programs, LINGUA FRANCA offers two distinct coordination strategies [16]. The default coordinator is a centralized component called the RTI (for runtime infrastructure) that guarantees that events are processed in the proper semantic order regardless of communication delays, computation time, or clock synchronization. An alternative coordinator is decentralized and guarantees proper semantic order only under clearly stated assumed bounds on communication latency, computation times, and clock synchronization error. This coordinator also provides hooks to handle faults that occur when these assumptions are violated. Deadline statements in LF can also be viewed as clearly stated assumptions coupled with fault handlers to deal with violations of those assumptions.

In short, LINGUA FRANCA offers deterministic behavior under clearly stated assumptions, mechanisms to detect when these assumptions are violated, and fault handlers so that applications can react appropriately to violation of the assumptions. This determinism applies even with parallel and distributed execution of LF programs, bringing the key advantages of determinism [104]: repeatability, consensus, predictability (sometimes), fault detection, simplicity, unsurprising behavior, and composability. For applications that require (or benefit from) nondeterminism, LF includes explicitly nondeterministic constructs that

```

1 reactiveclass Cockpit(10) {
2   knownrebecs {
3     Door door;
4     Vision Vision;
5   }
6   statevars { boolean open_value; }
7   Cockpit() {
8     // Schedule a physical action.
9     self.reaction_1();
10  }
11  @globalPriority(2)
12  msgsrv reaction_1() {
13    open_value = true;
14    door.read_port_open(open_value) after(0);
15    Vision.read_port_trigger(open_value)after(0);
16  }
17 }
18 reactiveclass Vision(10) {
19   knownrebecs { Door door; }
20   statevars {
21     boolean ramp_value;
22     boolean trigger_value;
23     boolean trigger_is_present;
24   }
25   Vision() {
26     trigger_is_present = false;
27   }
28   @globalPriority(4)
29   msgsrv reaction_1() {
30     // Assume a ramp is detected
31     ramp_value = true;
32     door.read_port_disarm(ramp_value) after(0);
33   }
34   @globalPriority(3)
35   msgsrv read_port_trigger(boolean
36     _trigger_value) {
37     trigger_value = _trigger_value;
38     trigger_is_present = true;
39     self.reaction_1();
40   }
41 }
42
43 reactiveclass Door(10) {
44   statevars {
45     int status;
46     boolean open_value;
47     boolean open_is_present;
48     boolean disarm_value;
49     boolean disarm_is_present;
50   }
51   Door() {
52     status = 0;
53     open_is_present = false;
54     disarm_is_present = false;
55   }
56   @globalPriority(6)
57   msgsrv reaction_1() {
58     if (status == 0) status = 1;
59     disarm_is_present = false;
60   }
61   @globalPriority(8)
62   msgsrv reaction_2() {
63     // Deploy emergency slides.
64     if (status == 0) status = 3;
65     else if (status == 1) {
66       status = 2;
67     }
68     open_is_present = false;
69   }
70   @globalPriority(7)
71   msgsrv read_port_open
72   (boolean _open_value) {
73     open_value = _open_value;
74     open_is_present = true;
75     self.reaction_2();
76   }
77   @globalPriority(5)
78   msgsrv read_port_disarm
79   (boolean _disarm_value) {
80     disarm_value = _disarm_value;
81     disarm_is_present = true;
82     self.reaction_1();
83   }
84 }
85
86 main {
87   Cockpit c(d, v):();
88   Vision v(d):();
89   Door d():();
90 }

```

Figure 3.7: TIMED REBECA model encoding the semantics of the LINGUA FRANCA model.

can be used. This is a notable contrast with most other concurrent and distributed computing frameworks, which give you nondeterminism by default and leave it to the designer to build deterministic behavior when needed.

3.5 Verifying an Engineering Model using TIMED REBECA

In this section, we show how to use TIMED REBECA to verify an engineering model of

the aircraft door system, rather than a model mixing engineering requirements and scientific assumptions. Fig. 3.7 shows a TIMED REBECA model encoding the semantics of the LINGUA FRANCA program shown in Fig. 3.6. Strictly speaking, this TIMED REBECA model encodes the behavior of the LF program at the *instant* when the Cockpit component schedules the physical action **command**. We discuss the challenge of environmental modeling in Section 3.6.

Since the LF program is an engineering model, we interpret all reaction dependencies and temporal semantics as *requirements* on the system design. Therefore, in the TIMED REBECA model, we ensure that the same requirements hold for all message servers that represent LF reactions. In the LF model, when a physical action **command** is scheduled, the **disarm** and **open** input ports of Door should receive messages logically simultaneously, despite the fact that the Vision component sits between the **open** port and the **disarm** port. This temporal requirement necessitates that the Vision reaction must be invoked before both reactions in Door, regardless of actual network delays and execution times.

In addition, to preserve determinism, LF requires that all reactions within the same reactor execute in the order of their declarations. This means that when reactions in the Door component are triggered logically simultaneously, the reaction labeled 1 must execute before the reaction labeled 2, as explained in Section 3.4. Therefore, based on the requirements above, LF defines a unique correct sequence of reaction invocations upon the arrival of a physical action:

Cockpit reaction 2 $\rightarrow$ Vision reaction 1 $\rightarrow$ Door reaction 1 $\rightarrow$ Door reaction 2.

A correctly implemented LF runtime must deliver this sequence during program execution, and ensuring that an LF runtime is implemented correctly is nontrivial. When formally verifying the engineering model, however, the LF semantics can be *assumed*, which simplifies verification significantly.

To encode this well-defined sequence in TIMED REBECA, we use the **@globalPriority** annotation, which gives a TIMED REBECA message server a priority value that is respected across reactive classes. These priorities provide partial ordering constraints that, according to LINGUA FRANCA semantics, allow parallel execution while respecting dependencies. This is suitable in our use case, as LF’s requirements on the order of reaction invocations span multiple reactors. We give the four reactions in the sequence global priorities of 2, 4, 6, and 8 respectively (lines 11, 27, 53, 57 in Fig. 3.7). We set a reaction’s priority to twice its invocation order so that auxiliary handlers (lines 34, 65, 71), which encode the semantics of LF ports, can execute before the reactions they trigger.

We verify this engineering model in Afra, which confirms that **status** can never be 3, even under all possible network latencies and execution times. As long as the implementation correctly follows the LF semantics, we can fully trust this verification result.

3.6 Challenges

One challenge that arises is that, in order to perform verification, TIMED REBECA programs must usually include a model of the environment in which they operate. A verifiable TR system is closed. All inputs to the designed part of the system have to be modeled. They can be modeled nondeterministically using TIMED REBECA, but in order for bounded model checking to be tractable, the number of nondeterministic choices must be limited. We avoided this problem in the previous section by using a degenerate environment model, where the environment makes exactly one request for service. For many cases, this will not be sufficient. For example, it may be important to consider the rate at which requests for service can occur. These rates can be bounded in LINGUA FRANCA, which may make the modeling easier, but this remains to be explored.

Since we cannot design the environment in which an engineered system runs, a model of the environment is necessarily a scientific model, not an engineering model. The model can make *assumptions*, enabling a kind of “assume-guarantee” reasoning, but the model cannot be a *specification* of the environment. There is an art, therefore, to constructing models of the environment. Ideally, each such model represents the weakest possible assumptions such that verification remains tractable and the assumptions can reasonably be assumed to hold [167, 117]. In practice, patterns of environment behavior will need to be verified separately, because any single model with weak enough assumptions to admit all possible behaviors of the environment will lead to intractable verification problems.

3.7 Future Work

We have observed that models used for verification in practice sometimes conflate engineering models and scientific models, and that there is significant risk that this confusion compromises the trustworthiness of the verification results. We argue that systems under design should be modeled with engineering models, which are specifications of correct behavior, rather than scientific models, which approximate implementations. When this distinction is maintained, formal verification can definitively show whether the design meets those obligations. It is a separable problem to determine whether an implementation correctly realizes the specification. A key limitation, however, is that designs can only be verified under assumptions about the environment in which they operate, and that models of the environment are inevitably scientific models, not engineering models.

We have shown that LINGUA FRANCA programs can serve as effective engineering models for a deterministic version of actor networks, and that TIMED REBECA can be used to verify these models. Using TIMED REBECA’s ability to model nondeterminism, scientific models of the operating environment can be combined with the engineering models of the design, yielding verification results that are trustworthy under clearly stated assumptions about the operating environment of the designed system.

Chapter 4

Formal Verification of LINGUA FRANCA Programs

“Program testing can be used to show the presence of bugs, but never to show their absence!”

—Edsger W. Dijkstra, *“Notes on Structured Programming,”* 1970.

Cyber-physical systems (CPS) [103, 156, 116] play a major role in today’s society. From sensor networks to autonomous vehicles, from industrial automation to avionics, these systems have permeated every aspect of our daily lives. CPS applications are often safety-critical; needless to say, it is crucial that such systems are designed and implemented with assurance of correctness.

CPS design and verification can be challenging because one needs to capture the interactions between the digital world and the physical world, and these two worlds have properties that cannot be unified easily [103]. For example, the C language, a mainstream programming language for embedded applications, adopts a sequential programming model, whereas the physical world is intrinsically concurrent. In addition, *timing* is mostly considered as a side effect of the software implementation and most languages do not include time in their syntax or semantics; however, time is central to the evolution of physical processes and timing should be treated as a correctness criterion rather than a performance metric. A CPS-oriented technology stack should address both concurrency and timing, which should also be addressed in CPS formal models.

CPS verification needs more automation. Although advances in model checking and automated theorem proving, such as bounded model checking (BMC) [22, 21] and satisfiability modulo theory (SMT) solvers [14], have gained popularity due to their ability to help automate verification, verification models are often generated by hand. This can be very time-consuming for the highly complex systems typical in CPS [103]. It is important to lower the burden of applying formal methods by *fully automating* verification, which requires not only automated verification algorithms but also generating verification models

automatically from the systems under verification, with an interface that is easy to use and tightly integrated with design and development. In the ideal case, the language used for design should also naturally support the verification process.

One natural formalism for CPS modeling is that of discrete-event systems (DES) [34]. However, formally modeling the behavior of DES using a conventional operational model (*i.e.*, a transition system) is difficult because one needs to specify properties globally over execution traces, a task that cannot be easily achieved at the transition level. In addition, modeling concurrent execution often involves modeling partial order traces, which typically involves the use of nondeterminism. Therefore, when it comes to modeling real-time, concurrent CPS as DES, one needs to look for a modeling strategy more suitable than the operational approach. In this chapter, we focus on building *axiomatic models*.

Our solution. Our solution for building verifiable CPS starts with the adoption of LINGUA FRANCA (LF) [129], a language suitable for modeling and implementing CPS applications. LINGUA FRANCA is a polyglot coordination language designed to have native support for time and concurrency and to adopt a discrete-event semantics. LF endows popular programming languages like C (which we focus on in this chapter), C++, and Rust with mechanisms to define and compose reactive and concurrent software components called *reactors*. While traditional CPS implementations in low-level languages (*e.g.*, C) expose no easy way of extracting the interactions between subsystems, LF programs provide highly analyzable program structures and a well-defined deterministic semantics, exposing the interactions between subsystems clearly and explicitly. These features make it well-suited for formal verification. LF also blurs the boundary between *models* and *implementations* by allowing the user to specify both the software and the environment. LF has been used to implement real-life applications, including Autoware [16], AUTOSAR [138], distributed databases [111], and labs in embedded systems courses [81].

In this chapter, we aim to ensure the functional correctness of LF programs and thus define a verification problem of checking Safety Metric Temporal Logic (Safety MTL) formulas [148] for LF programs. To perform verification, we present a methodology to encode the execution of an LF program as an *axiomatic* model, instead of an operational model that aims to build an abstraction of the system using explicit notions of states and transitions [6, 63]. Axiomatic models allow us to specify LF behavior based on a well-defined LF *semantics*. An axiomatic model can thus be sound w.r.t. *all* valid runtime implementations that correctly implement the same LF semantics. This is a significant advantage over operational models, which need to be carefully designed to be sound abstractions of a subset of the possible implementations. To make BMC complete, we propose an algorithm that calculates an upper bound on the BMC *completeness threshold* based on an LF program under verification and a Safety MTL property.

We implement our approach in the LF VERIFIER, which can verify an LF program *fully automatically*. For each LF implementation, the LF VERIFIER *generates* a verification model for UCLID5 [171, 152], a modeling and verification framework. The UCLID5 model is com-

piled into an SMT query and then checked automatically by the Z3 solver [43]. The UCLID5 model generation is enabled by the program structure exposed by LF and its deterministic semantics.

Related work. Outside of LINGUA FRANCA, our work is influenced by the existing literature on verifying dataflow languages [64, 65, 55, 47, 189], Petri nets [49, 53], actors [161, 191, 100, 13], and other SMT-based techniques [6, 142]. The UCLID5 verifier we use is an evolution of the older UCLID system [27], one of the first SMT solvers and SMT-based verifiers. In terms of the underlying encoding and the verification strategies, we note that various works [153, 64, 142, 132] implement similar axiomatic models in other domains.

Within the scope of LINGUA FRANCA, we are aware of two works that are directly related to verifying reactor systems at the time of this writing. Sirjani et al. [176] map LF programs onto models written in the TIMED REBECA language [158] and perform model checking using the REBECA model checker [77], an explicit-state model checker. They model the reactor systems based on the event-based semantics and show that their approach can handle *state* properties. Deantoni et al. [44] integrate the MoCCML [45] into LF’s Eclipse IDE and leverage the CADP toolbox [57] to check CCSL [9] properties, which can be considered a variant of *dataflow* properties. We will expand on the difference between state and dataflow properties in Sec. 4.3.

Our work differs from these prior approaches significantly:

1. LF VERIFIER is the first fully automated verifier for LINGUA FRANCA and generates verification models directly from LF programs. Existing approaches require the user to first manually translate an LF program into an analyzable representation.
2. Our approach supports a well-defined fragment of Safety MTL, a temporal logic well-suited for real-time systems.
3. Our tooling uses an intermediate UCLID5 encoding that is much easier to manipulate and matches the LF semantics precisely.

Contributions. This chapter makes the following key contributions:

- We advocate for the use of LINGUA FRANCA for a “design-for-verification” approach to CPS design. We define a verification problem for checking Safety MTL properties of LF programs.
- We present an approach to performing BMC on axiomatic models of LF programs, which models executions of LF programs at the semantics level and hence ensures soundness for all valid LF runtime implementations. To ensure the completeness of BMC, we present an algorithmic method to upper bound the completeness threshold based on an LF program and a Safety MTL property.

- We present the LF VERIFIER, which can automatically *generate* axiomatic models directly from an LF program. The models are subsequently checked by the UCLID5 verification engine with the Z3 solver. The LF VERIFIER also provides an annotation system fully integrated into the LF development environment.
- We developed a benchmark suite that consists of 22 LF programs sampled from real-life applications and benchmarks for Erlang, Lustre, actor-oriented languages, and RTOSes, and we evaluate the LF VERIFIER against the benchmark suite. The results show that the LF VERIFIER correctly checks 21 out of 22 programs automatically.

4.1 Running Example: ADAS

The reactor model [126, 123] is a deterministic model of concurrent computation for reactive systems. It adopts a discrete-event semantics and extends the actor model [72] with a deterministic order of handling concurrent timestamped messages. Reactors also share similarities with synchronous languages [17], logical execution time (LET) [68], dataflow [46, 143], and process networks [82].

To introduce the reactor model informally, we present a simplified model of an advanced driver assistance system (ADAS) [196], commonly found in modern vehicles. In this system, a camera and a LiDAR periodically collect data from the environment and send them to an ADAS processor, which performs image recognition on the given data. If an object is getting too close to the vehicle, a warning message is displayed on the dashboard. If the object is within some safety-critical distance, then the ADAS processor automatically applies the brakes. The user can also manually apply the brakes by pressing the pedal.

We now explain how the diagram in Fig. 4.1 represents this simplified ADAS system. A *reactor* is a stateful concurrent component. In the diagram, reactors are represented as rectangles with rounded corners, and they are **Camera**, **LiDAR**, **Pedal**, **Brake**, **Dashboard**, **ADASProcessor**, and **ADASModel**. A reactor can contain *reactions*, *actions*, *timers*, *ports*, and *connections*.

A reaction (*e.g.*, line 6) is a routine that executes when any of its triggers is present, and it is rendered as a chevron in the diagram. The body of a reaction (*e.g.*, lines 7-9) is written in a *target* language and represents the “business logic” of the application. In this example, the target language is C (line 1), and C code snippets are wrapped between the {= and =} brackets. In addition, the **ADASProcessor** reactor (lines 37-54) has two reactions, the one on the left labeled 1 and the one on the right labeled 2. These numbers correspond to the *priorities* of the reactions within **ADASProcessor**: when they are triggered simultaneously, reaction 1 will fire before reaction 2. It is important to note that reaction priorities establish a total order only within the same reactor.

A reaction can be triggered by a port (*e.g.*, line 38), a timer (*e.g.*, line 5), or an action (*e.g.*, line 42). A port, rendered as a black triangle in the diagram, is used to communicate with other reactors. A timer, rendered as a clock in the diagram, is used to generate periodic

```

1 target C
2 reactor Camera {
3   output out :{=c_frame_t=}
4   state frame:={c_frame_t=}
5   timer t(0,17ms) //60fps
6   reaction(t) -> out {=
7     // Capture a frame.
8     self->frame = ...;
9     lf_set(out,self->frame);
10  =}
11 }
12 reactor LiDAR {
13   output out :{=l_frame_t=}
14   state frame:={l_frame_t=}
15   timer t(0,34ms) //30fps
16   reaction(t) -> out {=
17     // Capture a frame.
18     self->frame = ...;
19     lf_set(out,self->frame);
20  =}
21 }
22 reactor Pedal {
23   output out:int
24   physical action a
25   reaction(a) -> out {=
26     lf_set(out, 1); =}
27 }
28 reactor Brakes {
29   input inADAS:int
30   input inPedal:int
31   state brakesApplied:int(0)
32   reaction(inADAS,inPedal){=
33     // Actuate brakes.
34     self->brakesApplied = 1;
35     =} deadline(10ms) {=.}=}
36 }
37 reactor ADASProcessor {
38   input in1:={c_frame_t=}
39   input in2:={l_frame_t=}
40   output out1:int
41   output out2:int
42   logical action a(50ms)
43   state requestStop:int
44   reaction(in1,in2) -> a {=
45     // ... Detect danger
46     // and request stop.
47     lf_schedule(a, 0);
48     self->requestStop = 1;
49   =}
50   reaction(a) -> out1,out2{=
51     if (self->requestStop==1)
52       lf_set(out1, 1);
53   =} deadline(20ms) {=.}=}
54 }
55 reactor Dashboard {
56   input in:int
57   state received:int
58   reaction(in) {=
59     self->received = 1; =}
60 }
61
62 @property(name="responsive",
63 tactic="bmc", spec="G[0,10 ms]
64 ((ADASModel_l_reaction_0 &&
65 (F[0](ADASModel_p_requestStop
66 ==1)))==>(F[0,55 ms](ADASModel
67 _b_brakesApplied==1))))")
68 main reactor ADASModel {
69   c = new Camera()
70   l = new LiDAR()
71   v = new ADASProcessor()
72   b = new Brakes()
73   d = new Dashboard()
74   p = new Pedal()
75   l.out -> v.in1
76   c.out -> v.in2
77   v.out2 -> d.in
78   v.out1 -> b.inADAS after 5ms
79   p.out -> b.inPedal
80 }

```

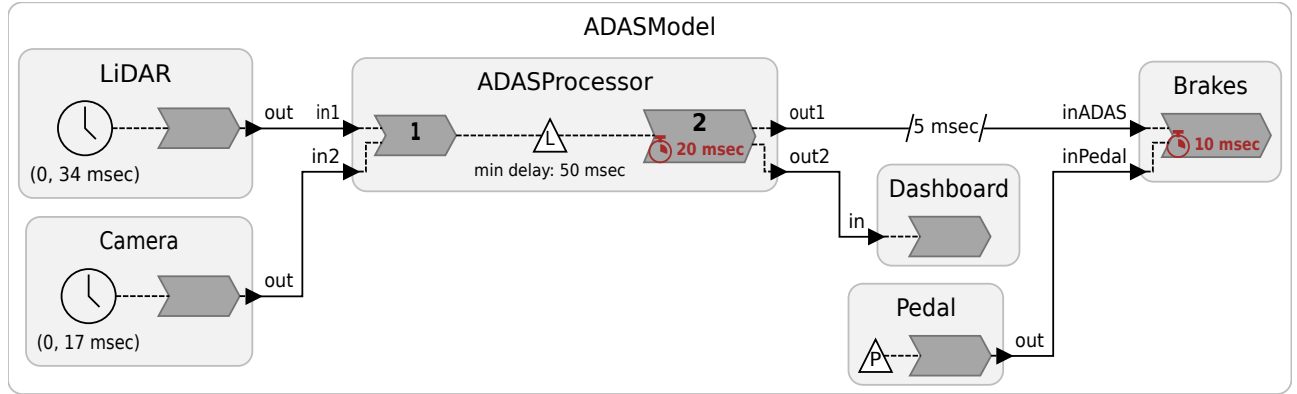


Figure 4.1: LF code and diagram of a simplified model of an ADAS system.

signals. An action is used to schedule future events within a reactor. It is rendered as a white triangle with an “L” if it is a *logical* action, or a “P” if it is a *physical* action. A logical action is scheduled by a reaction, such as reaction 2 in **ADASProcessor**. A physical action is scheduled asynchronously by the environment, such as a press on the brake pedal that triggers an interrupt service routine (ISR). Connections can be drawn between two ports using connection statements such as lines 75-79. Logical connections are specified using `->` with an optional logical delay specified using `after` (e.g., line 78). Without a delay specified, an event travels from an output port to an input port logically instantaneously.

All events in the system are handled in the order of *tags*, which use a superdense repre-

sensation of time [135], encoded as tuples of the form $g = (t, m)$, where $t \in \mathbb{T}$ is the *time value* and $m \in \mathbb{N}$ is a *microstep index*. Reactions are logically instantaneous; *logical time* does not elapse during the execution of a reaction (*physical time*, denoted as T , on the other hand, does elapse). If a reaction produces an output that triggers another reaction, then the two reactions execute logically simultaneously (*i.e.*, at the same tag). Deadlines can be specified and are violated when $T - t$ is greater than the specified intervals (*e.g.*, lines 35 & 53).

LF features used in this work. The formal analyses from the proposed methodology operate on the *logical* behavior of the program, and the set of language features this work addresses includes all features introduced earlier except *physical* actions, deadlines, and advanced features such as modal models [164]. LINGUA FRANCA is a highly expressive language, and our proposed methodology makes use of most but not all of the language features available. We leave analyzing programs with these language features for future work.

Safety properties in reactors. Safety properties are properties that state, in loose terms, “something bad should not happen.” We focus on safety properties because they are generally considered the most important class of properties for safety-critical systems, and they tend to be amenable to automated techniques [65]. In addition, since bounded liveness properties can be converted into safety properties [19], we do not make a distinction between bounded liveness and safety properties.

In reactors, we identify three types of properties: (I) *state* properties, (II) *dataflow* properties, and (III) *mixed* properties. State properties focus on the values of *variables*. In the context of reactors, they include state variables, ports, and actions. Dataflow properties, on the other hand, specify the temporal behavior of *reaction invocations*. Mixed properties combine state and dataflow properties.

To illustrate the difference between these types of properties, consider the **ADASModel** reactor shown in Fig. 4.1. The property “*The ‘frame’ variable in Camera is initially set to an empty value*” is a state property (Type I) that only refers to the state variable *frame*. The property “*In ADASProcessor, Reaction 2 is invoked within 100 ms after Reaction 1 is invoked.*” is a dataflow property (Type II). Notice that this property only refers to reaction invocations and does not refer to variables. Lastly, a mixed property (Type III) is shown at lines 62-67 in the LF program in Fig. 4.1. In natural language, the property states “*Globally from time 0 to 10 ms, when the LiDAR reaction invokes and the ADASProcessor requests stop at the same tag, the brakes will eventually be applied within 55 ms.*” We write the property in MTL

```
1 G[0, 10 ms]((ADASModel_l_reaction_0 && (F[0](ADASModel_p_requestStop == 1)))
2 ==> (F[0, 55 ms](ADASModel_b_brakesApplied == 1)))
```

where **ADASModel_l_reaction_0** refers to the invocation of the reaction inside the **LiDAR** reactor, **ADASModel_p_requestStop** is the **requestStop** variable inside the **ADASProcessor** reactor, and

`ADASModel_b_brakesApplied` is the `brakesApplied` variable inside the `Brakes` reactor. We wrap the property inside an `@property` annotation, which we will expand on and verify in Sec. 4.5.

At a high level, the key contribution of this chapter is to give a methodology for verification where the only thing a designer needs to do is add a specification of the safety requirements (as an `@property` annotation) to the *implementation* program. This contrasts with most verification methodologies, which require the designer to first construct a *model* of the implementation. Here, we work directly with the implementation, which is given in LINGUA FRANCA. The model of the implementation is generated automatically from the LINGUA FRANCA program, as is the low-level C code that realizes the application.

4.2 Formal Logic and Properties

Metric Temporal Logic. Metric Temporal Logic (MTL) [91], a real-time extension of Linear Temporal Logic (LTL), has been a successful logic for specifying properties of reactive, embedded systems. The syntax and semantics of MTL are given below. We cite the definitions of MTL from Ouaknine and Worrell [148, 147]. The syntax and semantics of MTL are given as follows.

Definition 1 (Syntax of MTL). *Given an alphabet Σ of events, the formulas of MTL are built up from Σ by Boolean connectives and time-constrained versions of the next operator $\mathbf{X}$, the until operator $\mathbf{U}$, and the dual until operator $\tilde{\mathbf{U}}$ as follows:*

$$\psi ::= \text{true} \mid \text{false} \mid a \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \mathbf{X}_I \psi \mid \psi_1 \mathbf{U}_I \psi_2 \mid \psi_1 \tilde{\mathbf{U}}_I \psi_2$$

where $a \in \Sigma$, and $I \subseteq \mathbb{R}_{\geq 0}$ is an open, closed, or half-open interval with end points in $\mathbb{R}_{\geq 0} \cup \{\infty\}$.

Definition 2 (Semantics of MTL). *Given a (finite or infinite) time word $\pi = (\sigma, \tau)$ over alphabet Σ , a word position $i < |\pi|$, and an MTL formula ψ , the satisfaction relation $(\pi, i) \models \psi$ (read π satisfies ψ at position i) is defined as follows.*

$$\begin{aligned} (\pi, i) \models a & \quad \text{iff} \quad \sigma_i = a \\ (\pi, i) \models \psi_1 \wedge \psi_2 & \quad \text{iff} \quad (\pi, i) \models \psi_1 \wedge (\pi, i) \models \psi_2 \\ (\pi, i) \models \psi_1 \vee \psi_2 & \quad \text{iff} \quad (\pi, i) \models \psi_1 \vee (\pi, i) \models \psi_2 \\ (\pi, i) \models \mathbf{X}_I \psi & \quad \text{iff} \quad i < |\pi| \wedge \tau_{i+1} - \tau_i \in I \wedge (\pi, i+1) \models \psi \\ (\pi, i) \models \psi_1 \mathbf{U}_I \psi_2 & \quad \text{iff} \quad \exists j, i \leq j \leq |\pi| \wedge \tau_j - \tau_i \in I. [(\pi, j) \models \psi_2 \\ & \quad \wedge \forall k, i \leq k < j. (\pi, k) \models \psi_1] \\ (\pi, i) \models \psi_1 \tilde{\mathbf{U}}_I \psi_2 & \quad \text{iff} \quad \forall j, i \leq j \leq |\pi| \wedge \tau_j - \tau_i \in I. [(\pi, j) \models \psi_2 \\ & \quad \vee \exists k, i \leq k < j. (\pi, k) \models \psi_1] \end{aligned}$$

From here, additional temporal operators can be defined: *constrained eventually* $\mathbf{F}_I$ and *constrained globally* $\mathbf{G}_I$, where $\mathbf{F}_I\psi = \mathbf{true}\mathbf{U}_I\psi$, and $\mathbf{G}_I\psi = \mathbf{false}\tilde{\mathbf{U}}_I\psi$. In this work, we focus on a fragment of MTL called *Safety MTL* [148, 146, 149], which requires the interval I of the until operator $\mathbf{U}_I$ to have finite length. Both the satisfiability problem and the model-checking problem for Safety MTL over infinite timed words are decidable. Within Safety MTL, in this work we additionally constrain the interval I of the next operator $\mathbf{X}_I$ and the dual until operator $\tilde{\mathbf{U}}_I$ to have finite length. We leave the relaxation of this constraint for future work.

Bounded Model Checking (BMC) BMC [22] is a well-known technique for bug finding and bounded verification. A model checker implementing BMC usually proceeds as follows: It first checks if a property ψ is valid along a path π for some k steps (denoted as $\pi \models_k \psi$). If a satisfying assignment is found at this point, the counterexample is returned to the user. If not, the model checker increments k and checks $\pi \models_{k+1} \psi$. The procedure continues until checking k steps becomes intractable, or a *completeness threshold* is reached. A completeness threshold [92], denoted as $\mathcal{CT}$, is a minimum number of k such that $\pi \models_{\mathcal{CT}} \psi \implies \pi \models \psi$, *i.e.* if π satisfies ψ up to $\mathcal{CT}$ steps, then π satisfies the property ψ . In Sec. 4.4, we present an algorithm for upper-bounding $\mathcal{CT}$.

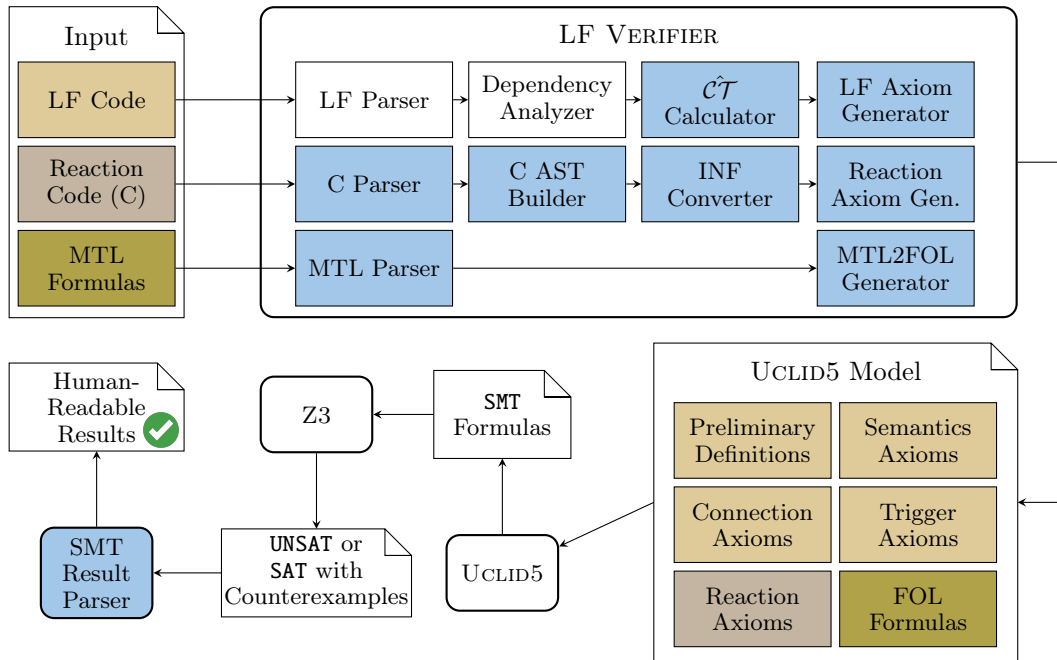


Figure 4.2: Verification workflow of LF programs and the architecture of the LF VERIFIER. Blue represents software modules implemented in this work. The other matching colors indicate correspondence between the input program and the generated model.

$$\begin{array}{lll}
\overline{indices} = (1, 2, \dots, k) & val \in vals = V^{|S|} & \pi_i = (rxn, tag, val, trig, sch, pld) \\
rxn \in rxns = N \cup \{\epsilon\} & trig \in trigs = \mathbb{B}^{|T|} & \bar{\pi} = (\pi_1, \pi_2, \dots, \pi_k) \in \Pi \\
t, m \in \mathbb{N} & sch \in schs = \mathbb{B}^{|A|} & indices = \overline{indices} \cup (k+1, \dots, k+|N|) \\
tag = (t, m) \in tags & pld \in plds = \mathbb{V}^{|A|} & \pi = \bar{\pi} \cup (\pi_{k+1}, \dots, \pi_{k+|N|}) \in \Pi
\end{array}$$

Figure 4.3: Logical time and path definitions.

4.3 Axiomatic Semantics for Verifying Reactors

Fig. 4.2 shows the verification workflow of LF programs and the architecture of the LF VERIFIER. In this section, we zoom in on the “UCLID5 model” in the figure and present the method of performing BMC using axiomatic models derived from the semantics of LINGUA FRANCA. We begin by describing the typical properties of interest in a system of reactors.

Axiomatic semantics. In this section, we give an axiomatic semantics for reactors and use the semantics to construct a model in UCLID5 to solve the verification problem defined below.

Definition 3 (Verification Problem). *Given a reactor, denoted as r , and a Safety MTL formula ψ , determine whether all execution paths $\bar{\pi}$ of r satisfy the formula ψ , i.e., $\bar{\pi} \models \psi$.*

In the proposed axiomatic semantics, a *reaction invocation* defines a transition, which matches the *event-based semantics* in Sirjani et al. [176]. To show the correspondence between the LF semantics in prose and our axiomatic semantics, we will constantly refer to parts of the LF semantics in prose from prior works [123, 126, 128] as we present the axiomatic semantics. The proposed semantics include four segments: (1) logical time and path; (2) reactor semantics; (3) connections and triggers; (4) reaction code. This section only includes the core axioms used to construct the axiomatic model.

(0) Notational conventions. We use uppercase letters to denote sets defined in LF (e.g. N is the set of reactions in an LF program) and italicized words to denote sets in an axiomatic model (e.g. $rxns$ is the set of reactions in a corresponding axiomatic model). A lowercase letter is always used to denote an element of a set (e.g. $n \in N$). For functions, we use the calligraphic font (e.g. $\mathcal{G}$) and italicized words (e.g. val_i). In addition, we use the blackboard-bold font to denote common sets (e.g. $\mathbb{N}$ for integers and $\mathbb{B}$ for $\{\mathbf{true}, \mathbf{false}\}$).

(1) Logical time and path. The definitions covered in this segment are shown in Fig. 4.3. A logical time tag is defined as $tag := (t, m)$, which is a tuple of two natural numbers: $t \in \mathbb{N}$

to represent a *timestamp* (i.e., the number of nanoseconds elapsed in logical time since the start of the program), and $m \in \mathbb{N}$ to represent a *microstep* index, which is used to enforce an ordering on events with the same timestamp. To extract a timestamp or a microstep from a tag, we define projection functions $\mathcal{T} : \text{tags} \rightarrow \mathbb{N}$ for mapping a time tag to its time value and $\mathcal{M} : \text{tags} \rightarrow \mathbb{N}$ for mapping a time tag to its microstep index. To compare two tags, we define an ordering relation

$$\begin{aligned} \prec = \{ & (tag_1, tag_2) \in \text{tags}^2 \mid \mathcal{T}(tag_1) < \mathcal{T}(tag_2) \\ & \vee (\mathcal{T}(tag_1) = \mathcal{T}(tag_2) \wedge \mathcal{M}(tag_1) < \mathcal{M}(tag_2)) \} \end{aligned}$$

and a relation $\preceq = \{(tag_1, tag_2) \in \text{tags}^2 \mid tag_1 \prec tag_2 \vee tag_1 = tag_2\}$.

We use R to denote the set of reactors and N the set of reactions defined in an LF program. In the axiomatic model, a reaction invocation is defined as $rxn \in rxns = N \cup \{\epsilon\}$, where ϵ indicates an empty reaction. Let S denote the union of *state variables*, *ports*, and *actions* defined in the LF program. A vector of values carried by elements in S is defined as $val \in vals = V^{|S|}$, where V is an abstract set of values. Let T denote the union of *ports* and *actions* defined in the LF program. A Boolean vector denoting the presence of triggers is defined as $trig \in trigs = \mathbb{B}^{|T|}$. We further let $A \subseteq T$ denote the set of *actions* specifically. A Boolean vector denoting whether actions are scheduled is defined as $sch \in schs = \mathbb{B}^{|A|}$. This is different from the same actions in *trigs*, which are set to true when the scheduled actions actually *become present* at a future time tag. When actions are scheduled, their payloads are stored in $pld \in plds = \mathbb{V}^{|A|}$.

A *path* $\bar{\pi} = (\pi_1, \pi_2, \dots, \pi_k)$ is a bounded sequence of states, where the length of the sequence is equal to a bound k . This is modeled in SMT using the theory of arrays with the constraint that any element outside of the range $[1, k]$ carries an empty value. We also define an *indices* set as a sequence of natural numbers from 1 to the bound k , i.e., $\overline{indices} = (1, 2, \dots, k)$. Intuitively, a path can be interpreted as an unrolled execution trace over which the axioms will be enforced. To ensure that certain axioms, to be introduced below, also hold at the boundary of the path (i.e., at step k), we define an extended version of the path with a *padding*, where $\overline{indices} = \overline{indices} \cup (k+1, \dots, k+|N|)$, $\pi = \bar{\pi} \cup (\pi_{k+1}, \dots, \pi_{k+|N|}) \in \Pi$, and the length of the padding is set to $|N|$. Note that Safety MTL formulas are evaluated on $\bar{\pi}$ and not on π , the padded version, which is used here to prevent the axioms from being trivially unsatisfiable at the path boundary.

A *state* $\in \text{states}$ is a 6-tuple $(rxn, tag, val, trig, sch, pld)$, where $rxn \in rxns$ is the reaction invoked, $tag \in \text{tags}$ is the logical time at which the invocation occurs, $val \in vals$ is the variable values after the invocation, $trig \in trigs$ the presence of triggers at the moment of the invocation, $sch \in schs$ indicates whether actions are scheduled during the current reaction invocation, and $pld \in plds$ the payloads actions carry when they are scheduled.

We use subscripts to denote an individual element in a vector. The i^{th} entry of val is denoted as val_i , and the i^{th} entry of $trig$ is denoted by $trig_i$. We use the same symbols and define projection functions $val_i : \text{states} \rightarrow V$ and $trig_i : \text{states} \rightarrow \mathbb{B}$, to extract the i^{th} value or trigger at a particular state in a path. We define a projection function $\mathcal{N} : \text{states} \rightarrow N$,

which extracts a *rxn* from a *state*, and a projection function $\mathcal{G} : \text{states} \rightarrow \text{tags}$, which extracts a *tag* from a *state*. We are now ready to start axiomatizing the core reactor semantics.

(2) Reactor semantics. One of the most important sets of axioms is the semantics of the reactor model. We first define an axiom enforcing that the ordering of reaction invocations must be in the order of time tags:

$$\forall i, j \in \text{indices}. \mathcal{G}(\pi_i) \prec \mathcal{G}(\pi_j) \implies i < j, \quad (\text{TimeOrder})$$

where $\prec$ is the ordering relation defined earlier. At a time tag, a reactor maintains a deterministic order of execution by executing its reactions in the order of their priorities.

$$\forall i, j \in \text{indices}. \mathcal{G}(\pi_i) = \mathcal{G}(\pi_j) \wedge \mathfrak{P}(\mathcal{N}(\pi_i)) < \mathfrak{P}(\mathcal{N}(\pi_j)) \implies i < j, \quad (\text{ReactionOrder})$$

where $\mathfrak{P} : N \rightarrow \mathbb{N}$ is a function that maps a reaction to a priority value, which is a natural number. Besides enforcing a tag order, we also require that a given reaction can only be invoked once at any logical time instant:

$$\forall i, j \in \text{indices}. (\mathcal{N}(\pi_i) = \mathcal{N}(\pi_j) \wedge i \neq j) \implies \mathcal{G}(\pi_i) \neq \mathcal{G}(\pi_j). \quad (\text{Uniqueness})$$

Next, we place constraints on the time tags, where both the timestamp and the microstep must be non-negative.

$$\forall i \in \text{indices}. \mathcal{T}(\mathcal{G}(\pi_i)) \geq 0 \wedge \mathcal{M}(\mathcal{G}(\pi_i)) \geq 0. \quad (\text{ValidTag})$$

For an empty invocation ϵ , we require that it must always occur at the end of the trace. In other words, ϵ invocations cannot occur in between real reaction invocations in a trace. This behavior is desired because the purpose of ϵ events is to fill in unused entries in the trace array when modeling an execution trace that is shorter than the verification bound.

$$\forall j \in \text{indices}. \mathcal{N}(\pi_j) \neq \epsilon \implies (\forall i \in \text{indices}, i < j. \mathcal{N}(\pi_i) \neq \epsilon) \quad (\text{EmptyLast})$$

(3) Connections and triggers. Since we represent the system axiomatically, we do not explicitly specify the transition relation between each state variable, but instead specify a set of invariants over the traces being examined.

Let p_u and p_d be a pair of upstream and downstream ports, with their values stored in $val_u, val_d \in \text{val}$ and their presence in $trig_u, trig_d \in \text{trig}$. We use a variable $physical \in \mathbb{B}$ to denote whether the connection is a *physical* connection, and we define a variable $delay \in \mathbb{N}$ to indicate the logical delay carried by the connection. We also define a *schedule* operator $\oplus : \text{tags} \times \mathbb{N} \rightarrow \text{tags}$ as $g \oplus t = g'$ s.t. $\mathcal{T}(g') = \mathcal{T}(g) + t$ and $\mathcal{M}(g') = 0$ if $t > 0$ and $\mathcal{M}(g') = \mathcal{M}(g) + 1$ if $t = 0$. The behavior of a connection is specified as follows.

$$\begin{aligned} \forall i \in \overline{\text{indices}}. (trig_u(\pi_i) \implies (\exists j \in \text{indices}. j > i \wedge trig_d(\pi_j) \\ \wedge val_d(\pi_j) = val_u(\pi_i) \wedge (\neg physical \implies \mathcal{G}(\pi_j) = \mathcal{G}(\pi_i) \oplus delay))) \quad (\text{SendToDownstream}) \end{aligned}$$

The SendToDownstream axiom specifies that when the upstream trigger is present at some step in the path, there exists a step in the path with the downstream trigger present. We further specify the Correspondence axiom below to ensure a one-to-one correspondence between the presence of triggers in logical time.

$$\begin{aligned} \forall i \in \text{indices}. (\text{trig}_d(\pi_i) \implies (\exists j \in \text{indices}. j < i \wedge \text{trig}_u(\pi_j) \\ \wedge \text{val}_u(\pi_j) = \text{val}_d(\pi_i) \wedge (\neg \text{physical} \implies \mathcal{G}(\pi_i) = \mathcal{G}(\pi_j) \oplus \text{delay}))) \quad (\text{Correspondence}) \end{aligned}$$

Since the default behavior of variables in an axiomatic model is nondeterministic assignment, the SameIfAbsent axiom states that when a trigger is absent, the value corresponding to the trigger stays the same as the previous step.

$$\forall i \in \text{indices}. \text{trig}_d(\pi_i) = \text{false} \implies (\text{val}_d(\pi_i) = \text{val}_d(\pi_{i-1})) \quad (\text{SameIfAbsent})$$

If a reaction $v \in N$ is sensitive to a startup trigger, then it will be invoked at tag $(0,0)$. To encode this behavior, we state a StartupTrigger axiom.

$$\exists i \in \text{indices}. \mathcal{N}(\pi_i) = v \wedge \mathcal{G}(\pi_i) = (0,0) \wedge \neg(\exists j \in \text{indices}. \mathcal{N}(\pi_j) = v \wedge j \neq i) \quad (\text{StartupTrigger})$$

Besides ports, LF also supports actions and timers as triggers. Let us first start with actions. To axiomatize the behavior of actions, we let a be an action with its value stored in $\text{val}_a \in \text{vals}$, its presence stored in $\text{trig}_a \in \text{trigs}$, and a logical delay $\text{delay}_a \in \mathbb{N}$. We further let reactions_a denote a set of reactions that can schedule the action a . We can state an axiom, ScheduleAction, to encode the semantics of actions.

$$\begin{aligned} \exists i \in \text{indices}. \bigvee_{n \in \text{reactions}_a} \text{trig}_a(\pi_i) = \text{true} \implies (\exists j \in \text{indices}. j < i \wedge \mathcal{N}(\pi_j) = n \\ \wedge \mathcal{G}(\pi_i) = \mathcal{G}(\pi_j) \oplus \text{delay}_a \wedge \text{sch}_a(\pi_j) = \text{true} \wedge \text{val}_a(\pi_i) = \text{pld}_a(\pi_j)) \quad (\text{ScheduleAction}) \end{aligned}$$

Timers are another important feature for generating periodic events. To axiomatize their behavior, let t be a timer with an initial offset $\text{offset} \in \mathbb{N}$, a period $\text{period} \in \mathbb{N}$, and its presence stored in $\text{trig}_t \in \text{trigs}$. Recall also that k is the largest index in $\overline{\text{indices}}$. Then, a timer must start firing at the predefined offset, as stated in TimerOffset.

$$\begin{aligned} ((\mathcal{T}(\mathcal{G}(\pi_k)) \geq \text{offset}) \implies \exists i \in \overline{\text{indices}}. \text{trig}_t(\pi_i) = \text{true} \wedge \mathcal{G}(\pi_i) = (\text{offset}, 0)) \\ \wedge ((\mathcal{T}(\mathcal{G}(\pi_k)) < \text{offset}) \implies \forall j \in \overline{\text{indices}}. \mathcal{N}(\pi_j) \neq \epsilon) \quad (\text{TimerOffset}) \end{aligned}$$

Intuitively, if the last state of $\bar{\pi}$ (i.e., π_k) has reached the timestamp specified by offset , then there must be a state in which the timer trigger is present and the timestamp matches offset . Otherwise, if the last state does not reach the timestamp specified by offset , all the reaction invocations along the path must be non-empty. The subsequent firing should occur

repeatedly at an interval specified by the period. We state two axioms, *TimerPeriod* and *TimerSpacing* to encode this behavior.

$$\begin{aligned} \forall i \in \overline{indices}. trig_t(\pi_i) = true &\implies \\ (\exists j \in indices. j > i \wedge trig_t(\pi_j) = true \wedge \mathcal{G}(\pi_j) = \mathcal{G}(\pi_i) \oplus period) &\quad (\text{TimerPeriod}) \end{aligned}$$

$$\begin{aligned} \forall i \in \overline{indices}. trig_t(\pi_i) = true &\implies \\ ((\exists x \in \mathbb{N}. x \geq 0 \wedge \mathcal{T}(\mathcal{G}(\pi_i)) = offset + x \times period) \wedge \mathcal{M}(\mathcal{G}(\pi_i)) = 0) &\quad (\text{TimerSpacing}) \end{aligned}$$

The LF semantics specifies that a reaction is triggered when any of its triggers is present. To encode this behavior, we state a *ReactionTriggers* axiom. Let $n \in N$ be a reaction and let $triggers_n$ be the set of triggers the reaction n is sensitive to.

$$\forall i \in indices. (\exists t \in triggers_n. t = true) \iff \mathcal{N}(\pi_i) = n \quad (\text{ReactionTriggers})$$

(4) Reaction code. To axiomatize reaction body code written in C, we adopt the preprocessing method presented by Armando et al. [10], in which a block of C code is transformed into a representation called *If Normal Form*. Starting from a C abstract syntax tree (AST) in *If Normal Form*, we then translate each if block into an implication where the if condition becomes the antecedent and the then branch becomes the consequent. Since axiomatizing C code is not our contribution, we will not expand on it here but will later revisit it in Sec. 4.5.

4.4 Bounding the Completeness Threshold

Recall that the *completeness threshold*, denoted as $\mathcal{CT}$, is an attribute of a system M with respect to a property p and a translation scheme such that checking a trace up to $\mathcal{CT}$ steps without errors proves $M \models p$ [21]. In the context of our work, the system M is a reactor system. The property p is a Safety MTL formula. The translation scheme is the translation of a reactor system to an axiomatic model (Sec. 4.3) and the translation of a Safety MTL formula to a First-Order Logic formula over a timed word $\bar{\pi}$ based on the semantics of MTL (Def. 2).

Here, we do not seek to compute the precise completeness threshold since such computation is as hard as the BMC problem itself [94]. Instead, we efficiently compute an upper *bound* on the completeness threshold, and we denote this bound as $\hat{\mathcal{CT}}$. Since we adopt the event-based semantics introduced by Sirjani et al. [176] (*i.e.*, modeling an LF execution as a sequence of reaction invocations), computing a bound for $\hat{\mathcal{CT}}$ requires finding the maximum number of reaction invocations within a logical time interval specified by the Safety MTL property. To do this, we construct a *state space diagram* for an LF program.

State space diagram. Intuitively, each node in the state space diagram consists of a time tag, a set of reactions invoked at this tag, and a set of *pending events* set to appear at some future time tags. The state space diagram of the running example is shown in Fig. 4.4.

Formally, a state space diagram is a directed graph (V, E) where $V \subseteq \text{tags} \times \mathcal{P}(\text{rxns}) \times \mathcal{P}(\text{events})$ and $E \subseteq V \times V$, where $\mathcal{P}$ denotes a power set. The symbols *tags* and *rxns* are defined previously in Fig. 4.3. Here we introduce a new set *events*, which is defined as $\text{events} = \text{tags} \times \text{trigs}$, where *trigs* is also defined in Fig. 4.3. To extract a tag, a set of reactions, or a set of events from a node, we define projection functions $\text{tag}_n : V \rightarrow \mathcal{P}(\text{events})$, $\text{rxns} : V \rightarrow \mathcal{P}(\text{rxns})$, and $\text{events} : V \rightarrow \mathcal{P}(\text{events})$. Similarly, to extract a tag or a trigger from an event, we define projection functions $\text{tag}_e : \text{events} \rightarrow \mathcal{P}(\text{events})$ and $\text{trigs} : \text{events} \rightarrow \mathcal{P}(\text{rxns})$.

Constructing a state space diagram requires a lightweight simulation of the *worst-case* execution until some time horizon, denoted as $\text{horizon} \in \mathbb{N}$, is reached, which is the time interval specified by a Safety MTL formula. We provide a precise definition of a theoretical worst-case execution below.

Definition 4 (Theoretical worst-case execution of an LF program). *A theoretical worst-case execution of an LF program is an execution path π permitted by the LF program structure such that*

$$\forall \pi' \in \Pi. \forall g \in \text{tags}. \left| \{ \pi_i \in \pi \mid \mathcal{G}(\pi_i) = g \wedge \mathcal{N}(\pi_i) \neq \epsilon \} \right| \geq \left| \{ \pi'_j \in \pi' \mid \mathcal{G}(\pi'_j) = g \wedge \mathcal{N}(\pi'_j) \neq \epsilon \} \right|$$

Intuitively, the theoretical worst-case execution path has the largest possible number of reaction invocations at each logical tag among all possible execution paths defined by the LF program structure.

During the simulation, the algorithm keeps track of three quantities (as defined earlier): a current time tag (denoted as *currentTag*), a list of reactions invoked at the current tag (denoted as *reactionsInvoked*), and a priority queue of pending events ordered by time tags (denoted as *pendingEvents*). The algorithm is listed in Algorithm 1 with specific implementation details abstracted away.

The main exploration logic occurs inside a while loop 5. Before the actual exploration begins, the algorithm requires a *horizon* argument and identifies a set of *initial events* from which the execution unfolds. The *horizon* argument is derived based on the given MTL formula as follows. Let ψ be an MTL formula in Negation Normal Form, and let $h : \text{MTL} \rightarrow \mathbb{R}$ be a function mapping an MTL formula to its horizon. The function h can then be defined recursively.

$$h(\psi) = \begin{cases} \max(h(\psi_1), h(\psi_2)) & \text{if } \psi \equiv \psi_1 \wedge \psi_2 \text{ or } \psi \equiv \psi_1 \vee \psi_2 \\ \max(I) + h(\psi_1) & \text{if } \psi \equiv \mathbf{G}_I \psi_1 \text{ or } \psi \equiv \mathbf{F}_I \psi_1 \text{ or } \psi \equiv \mathbf{X}_I \psi_1 \\ \max(I) + \max(h(\psi_1), h(\psi_2)) & \text{if } \psi \equiv \psi_1 \mathbf{U}_I \psi_2 \text{ or } \psi \equiv \psi_1 \tilde{\mathbf{U}}_I \psi_2 \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 1 Explore the state space and build a state space diagram

```

1: procedure EXPLORE(horizon, findLoop)
2:   previousTag, currentTag  $\leftarrow \epsilon$ , pendingEvents  $\leftarrow \emptyset$ 
3:   pendingEvents  $\leftarrow$  getInitialEvents()
4:   stop  $\leftarrow$  pendingEvents.isEmpty()
5:   while not stop do
6:     currentEvents  $\leftarrow$  processEarliestEvents(pendingEvents)
7:     reactionsInvoked  $\leftarrow$  triggerReactions(currentEvents)
8:     for each reaction in reactionsInvoked do
9:       pendingEvents  $\leftarrow$  pendingEvents  $\cup$  processReactionEffects(reaction)
10:    end for
11:    if previousTag = null then
12:      currentNode  $\leftarrow$  initializeCurrentNode()
13:    else if currentTag.timestamp > previousTag.timestamp then
14:      if findLoop and duplicateNodeExists() then
15:        identifyAndMarkLoop(); return
16:      end if
17:      newNode  $\leftarrow$  createNewNode(currentTag, reactionsInvoked, pendingEvents)
18:      addToDiagram(newNode)
19:      currentNode  $\leftarrow$  newNode
20:    else
21:      currentNode  $\leftarrow$  updateCurrentNode(reactionsInvoked, pendingEvents)
22:    end if
23:    previousTag  $\leftarrow$  currentTag
24:    currentTag  $\leftarrow$  pendingEvents.peek().tag
25:    if pendingEvents.isEmpty() or currentTag.timestamp > horizon.timestamp then
26:      stop  $\leftarrow$  true
27:    end if
28:  end while
29:  finalizeDiagram()
30: end procedure

```

To determine the set of initial events, the algorithm identifies the *startup* trigger used in the program and the initial firings of *timers* and adds them to *pendingEvents*.

Then, the procedure starts simulating the execution by stepping into a while loop, which can only be exited when the following stop conditions are met:

1. there are no more pending events (*i.e.*, $|pendingEvents| = 0$);
2. the simulation completes the time horizon (*i.e.*, $\mathcal{T}(currentTag) > horizon$);
3. a *cycle* is found in the state space diagram.

During each cycle iteration, the simulation pops all the pending events that are set to appear at the current tag off the *pendingEvents* queue. Then, reactions triggered by these events are added to the *reactionsInvoked* list. Note that since this is a lightweight simulation, the C bodies of these reactions are not simulated; instead, they are assumed to produce outputs at all the ports and schedule all the actions available to them. This is an over-approximation of the actual behavior and leads to the worst-case execution we seek. The set of new outputs and actions produced is inserted back into *pendingEvents*, waiting to be processed in some future loop iterations. When all the events at the current tag have been processed, the execution advances to the nearest future tag. A new node in the state space diagram is then created for the previous tag, storing the *reactionsInvoked* list and the *pendingEvents* queue. There is also an edge from the previous node to this new node.

We prove that the state space diagram generated by this algorithm is indeed the worst case.

Proposition 4.1. *Algorithm 1 produces a state space diagram representing the theoretical worst-case execution of an LF program.*

Proof. We prove this lemma by induction on the logical tag. In the base case, we focus on the first time tag and show that the algorithm computes the theoretical worst-case execution path up to the conclusion of the first time tag. When the algorithm begins, it adds initial events (line 3) including the startup trigger (set to occur at tag $(0, 0)$) and the initial firings of timers (which can occur at any time tag depending on the offsets) to *pendingEvents*. At this point, these initial events are not dependent on any prior state, but their addition to *pendingEvents* ensures that their effects will be considered when computing the next state, thus establishing state-dependency from the very beginning. If there are no initial events, the algorithm terminates and we arrive at a correct state space diagram that is empty. If there are initial events, we enter the while loop (line 5) and pop off the earliest events at line 6. Reactions triggered by these events are added to *reactionsInvoked* and **all** of their effects are added back to *pendingEvents*. This means that the effects of all the reactions invoked at the current state are taken into account when constructing the next state. These effects can influence both the state at the next time tag and the states at subsequent time tags, which ensures the propagation of state-dependency across different time tags. If the newly added effects happen at some future tag, we advance the current tag (line 24) and are

done with the first tag, leaving us with a correct state space diagram constructed up to the conclusion of the first tag with as many reactions invoked as possible at the first tag. On the other hand, if the added events are instantaneous, *e.g.*, setting output ports connected to a downstream input port with zero delay, then line 24 maintains the current tag, and the loop continues until we are done with the first tag. The resulting state space diagram is representative of the theoretical worst-case execution because during the loop iterations, all the possible reaction invocations have been accounted for. We thus conclude the base case.

For the inductive step, we make the inductive hypothesis that the algorithm has correctly computed a state space diagram up to some time tag later than the first tag; we seek to show that the algorithm can correctly compute the next tag. At this point, there must be unprocessed events in *pendingEvents* and the current tag must not have reached *horizon*, otherwise the algorithm would have been terminated at line 26. By our inductive hypothesis, *pendingEvents* contains the effects of all the reactions invoked at the previous states. Hence, the current state is dependent on all the previous states, and any reaction invoked at the current state will also take into account the effects of the reactions invoked at the previous states. We begin handling the next tag by popping the earliest events, checking the set of reactions invoked by these popped events, and again add **all** the effects of the invoked reactions back to *pendingEvents*. By adding all the effects of the reactions invoked at the current state to *pendingEvents*, we ensure that these effects are propagated to the next state and states at subsequent time tags. This maintains state-dependency across different time tags. If the newly added events are in the future, we update *currentTag* (line 24) and are done with the next tag. If not, line 24 does not update *currentTag*, and the loop continues until the next tag is finished, at which point, we are left with a state space diagram correctly constructed up to the conclusion of this next tag since all the possible reaction invocations have been accounted for. Invoking all possible reactions at each time tag results in the maximal number of reactions invoked over the entire execution, which constitutes the worst-case scenario. ■

Cycle detection. As an optimization, the simulation seeks to find a *cycle* in a state space diagram. LF programs often exhibit periodic behaviors (*e.g.*, the execution of our running example shown in Fig. 4.4). For such programs, the simulated execution does not need to continue until the completion of a *horizon*; it only needs to continue until the completion of a *period* (*i.e.*, a cycle). The rest of the execution up to the *horizon* can be inferred from the periodic behavior. To check whether an execution completes a period, when the execution advances to a new tag, the simulation checks whether the newly added node is *equivalent* to any of the previous nodes. Two nodes, n_1 and n_2 , are considered equivalent if

1. $rxns(n_1) = rxns(n_2)$;
2. $\{trigs(e_1) \subseteq trigs \mid e_1 \in events(n_1)\} = \{trigs(e_2) \subseteq trigs \mid e_2 \in events(n_2)\}$;

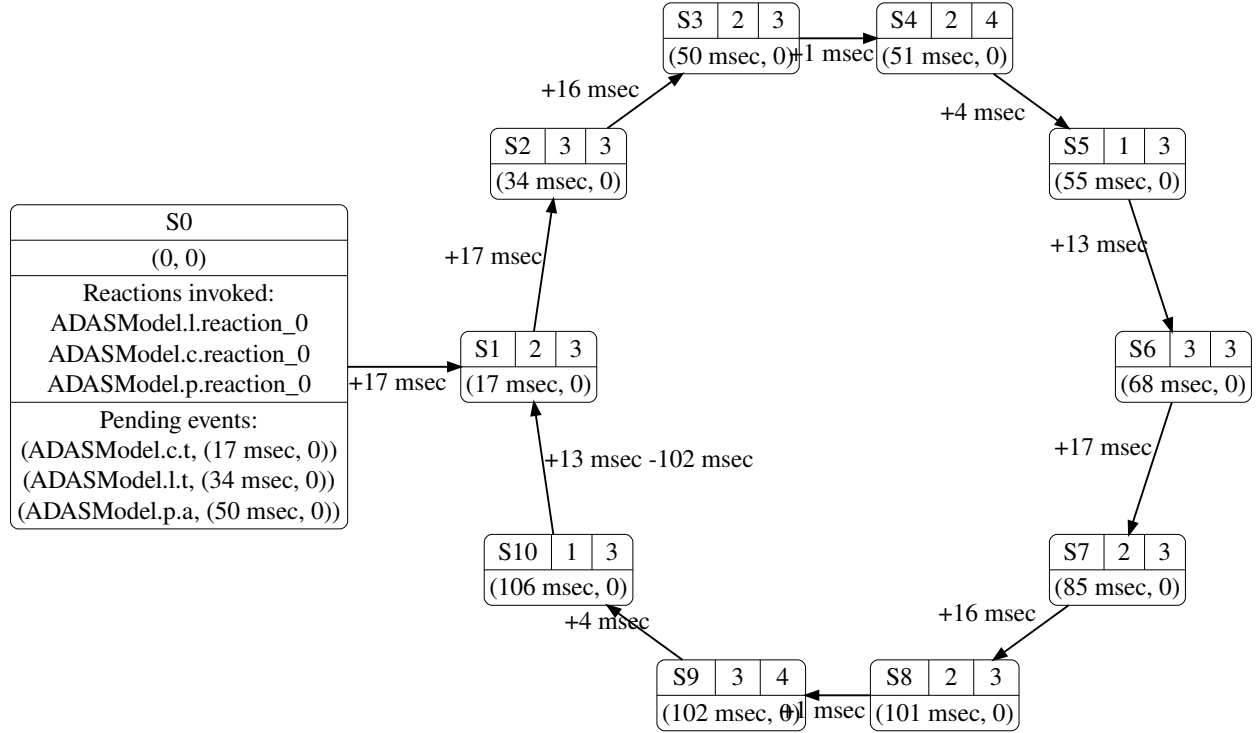


Figure 4.4: An abstract view of the state space diagram for the running example in Fig. 4.1. In this abstract view, except for S0, which is shown in full, each node consists of (from left to right) an identifier, the number of reactions invoked at the current tag, the number of events pending, and (at the bottom) the current tag.

$$3. \{tag_e(e_1) - tag_n(n_1) \mid e_1 \in events(n_1)\} = \{tag_e(e_2) - tag_n(n_2) \mid e_2 \in events(n_2)\}.$$

The cycle detection approach is similar to that of [60], which constructs state space diagrams for synchronous dataflow graphs [102].

From diagram to $\hat{\mathcal{CT}}$. Once the state space diagram is generated, we can compute $\hat{\mathcal{CT}}$ from it. For this, we state two theorems.

Theorem 1 ($\hat{\mathcal{CT}}$ w.r.t. a cycle-free diagram). *The completeness threshold w.r.t. an LF program that generates a cycle-free state space diagram (V, E) is upper bounded by $\hat{\mathcal{CT}}$ such that*

$$\hat{\mathcal{CT}} = \sum_{n \in V} |rxns(n)|.$$

Proof. A cycle-free diagram is by construction a finite linear sequence of nodes. For a node $n \in V$, the number of reaction invocations is given by $|rxns(n)|$. The maximum possible

number of reaction invocations, *i.e.*, $\hat{\mathcal{CT}}$, can be computed by summing up the number of reaction invocations of each node. We then obtain the desired expression. ■

Intuitively, this $\hat{\mathcal{CT}}$ counts the number of reaction invocations up to the completion of the horizon. If a cycle is found, we use its periodicity to infer $\hat{\mathcal{CT}}$.

Theorem 2 ($\hat{\mathcal{CT}}$ w.r.t. a diagram with a cycle). *The completeness threshold w.r.t. an LF program that generates a state space diagram (V, E) with a cycle is upper bounded by $\hat{\mathcal{CT}}$ such that*

$$\hat{\mathcal{CT}} = \sum_{n'' \in V'' \cup \{n_c\}} |rxns(n'')| + \sum_{n' \in V'} |rxns(n')| * iterations,$$

where V' denotes the set of nodes that form the cycle, $V'' = V \setminus V'$ denotes the rest of the nodes, $n_c \in V'$ denotes the node where the cycle begins, $iterations = \lceil (horizon - \mathcal{T}(tag_n(n_c))) / \rho \rceil$, and ρ denotes the cycle period.

Proof. A state space diagram with a cycle has an initialization phase, consisting of nodes $V'' \cup \{n_c\}$, and a periodic phase, consisting of nodes V' . Note that the first time of reaching $n_c \in V'$ counts toward the initialization phase since the periodic phase cannot start without first reaching n_c . The overall $\hat{\mathcal{CT}}$ is the sum of reaction invocations during both the initialization phase and the periodic phase. The first expression, $\sum_{n'' \in V'' \cup \{n_c\}} |rxns(n'')|$, follows directly from the proof of Theorem 1. The second expression over-approximates the number of reaction invocations during the periodic phase by multiplying the number of invocations during one iteration, given by $\sum_{n' \in V'} |rxns(n')|$ (also directly following the proof of Theorem 1), by $iterations$, the minimum number of cycle iterations required to guarantee completing the time horizon. The value of $iterations$ can be obtained by taking the remaining horizon after entering the periodic phase, *i.e.*, $horizon - \mathcal{T}(tag_n(n_c))$, dividing it by the period of the cycle, ρ , and applying the ceiling operation. We then obtain the desired expression. ■

4.5 LF VERIFIER: the First Fully Automated Verifier for Reactors

We now describe the implementation details of the LF VERIFIER, and walk through the steps to verify the safety property in the running example. The LF VERIFIER is embedded inside the LINGUA FRANCA compiler, which is open-source on GitHub¹. From the user's perspective, using the LF VERIFIER simply requires extending the original program with `@property` annotations; the rest is handled by invoking the compiler.

Recall from Fig. 4.2 that the user first provides an LF program along with annotations specifying temporal properties of the program in Safety MTL. Once the input files are provided, they are passed into the LF VERIFIER, which parses LF code, reaction code (written

¹GitHub link: <https://github.com/lf-lang/lingua-franca>

$ \begin{aligned} \textit{mtl} &::= \textit{formula} \mid \textit{unaryOp formula} \mid \\ &\quad \textit{formula binaryOp formula} \\ \textit{unaryOp} &::= '!' \mid 'X' \textit{interval} \mid \\ &\quad 'G' \textit{interval} \mid 'F' \textit{interval} \\ \textit{binaryOp} &::= 'U' \textit{interval} \\ \textit{interval} &::= ('(' \mid '[') \textit{time} ', ' \textit{time} (') \mid ']' \mid \\ &\quad '[' \textit{time} ']' \\ \textit{time} &::= \textit{int} (\textit{id})? \\ \textit{formula} &::= \textit{prop} \mid \textit{equivalence} \mid \textit{implication} \mid \\ &\quad \textit{disjunction} \mid \textit{conjunction} \mid \textit{mtl} \end{aligned} $	$ \begin{aligned} \textit{prop} &::= \text{'true'} \mid \text{'false'} \mid \\ &\quad \textit{expr relOp expr} \\ \textit{expr} &::= \textit{id} \mid \text{'(' sum ')'} \mid \textit{int} \mid \\ &\quad \textit{expr arithOp expr} \\ \textit{relOp} &::= '==' \mid '!=' \mid '<' \mid \\ &\quad '<=' \mid '>' \mid '>=' \\ \textit{arithOp} &::= '+' \mid '-' \mid '*' \mid '/' \\ \textit{equivalence} &::= \textit{mtl} '<==>' \textit{mtl} \\ \textit{implication} &::= \textit{mtl} '==>' \textit{mtl} \\ \textit{disjunction} &::= \textit{mtl} ' ' \textit{mtl} \\ \textit{conjunction} &::= \textit{mtl} '&&' \textit{mtl} \end{aligned} $
--	---

Figure 4.5: Syntax for the Safety MTL fragment supported by the LF VERIFIER.

in C, in our case), and the Safety MTL annotations. Based on the parsed LF code, LF VERIFIER analyzes the topology of the system and generates an axiomatic model that can be processed by UCLID5. In the rest of the section, we will focus on the modules used by the LF VERIFIER to verify properties and ignore the modules used for compiling the LF program.

Annotations. The user specifies an MTL property using an **@property** annotation. At the time of writing, the **@property** annotation requires three parameters: (i) the name of the property, (ii) the tactic used to check the property (only **bmc** is supported for now), and (iii) the MTL formula. A valid MTL formula follows the syntax shown in Fig. 4.5. Safety MTL is enforced by ensuring that “int” denotes finite non-negative integers. Reaction invocations (*resp.*, values of LF components) are specified using fully qualified names of the reactions (*resp.*, the components).

Parsing. After opening the LF file provided by the user, the LF VERIFIER parses the LF code using an “LF Parser” based on the Xtext framework [51, 20], and it parses the reaction body code (in C) and the MTL annotations using ANTLR4 parsers [151] (shown in Fig. 4.2 as “C Parser” and “MTL Parser”), generating a parse tree for C and another parse tree for MTL.

AST transformations. Once the C parse tree is generated by ANTLR4, the “C AST Builder” is used to build an abstract syntax tree (AST). Our implementation uses the technique presented by Armando et al. [10], which supports a subset of the C language. For

language features we do not currently support, including pointers, loops, and macros, the AST builder marks these AST nodes as *opaque* and returns a warning to the user. While C support is incomplete, the LF VERIFIER can currently analyze arithmetic operations, variable assignments, LF runtime APIs, and if statements (including nested if statements), which are arguably expressive enough to cover many real-time systems applications. We will discuss our plan to expand the support for C in Sec. 4.6.

Once the C AST is built, it is sent to the “INF Converter”, where INF stands for *If Normal Form*, a term introduced in Armando et al. [10]. The main idea is to flatten the nested if statements until an if condition is associated with each atomic statement. For example, the two reaction bodies in **ADASProcessor** in the running example (Fig. 4.1) effectively turn into

```
1 /* (C) Reaction body 1 */
2 if (true) lf_schedule(a, 0);
3 if (true) self->requestStop = 1;
4 /* (C) Reaction body 2 */
5 if (self->requestStop==1) lf_set(out1, 1);
```

The latter reaction body does not change since it follows the If Normal Form before transformation.

Computing $\hat{\mathcal{CT}}$. Let us return to the LF code. After the LF program passes through the “LF Parser”, it goes into a “Dependency Analyzer,” which extracts the dependency relations between reactions. When the dependency analysis is done, the LF VERIFIER invokes a “ $\hat{\mathcal{CT}}$ Calculator” which executes Algorithm 1 in Sec. 4.4 to obtain an upper bound on the completeness threshold.

To check the property in the running example, the LF VERIFIER first computes a logical time horizon from the property, which, in this case, is 65 ms: the outer $G[0, 10 \text{ ms}]$ operator requires 10 ms while the inner $F[0, 55 \text{ ms}]$ operator requires another 55 ms. The $F[0]$ does not contribute to the horizon due to the interval of 0. Therefore, to ensure the completeness of BMC, the model must unroll until 65 ms is reached.

The state space diagram of the program (Fig. 4.4) shows that within an interval of 65 ms, the program reaches six states (S0-S5). $\hat{\mathcal{CT}}$, *i.e.*, the maximum sum of reaction invocations along the six states, is $\hat{\mathcal{CT}} = 3 + 2 + 3 + 2 + 2 + 1 = 13$. The “ $\hat{\mathcal{CT}}$ Calculator” passes this number to the downstream model generators.

Model generation. During the model generation phase, three generators work together to produce a single UCLID5 model: an “LF Axiom Generator”, a “Reaction Axiom Generator”, and an “MTL2FOL Generator.” The “LF Axiom Generator” generates the definitions and axioms (except the reaction axioms) presented in Sec. 4.3. Here is a snippet of the generated UCLID5 code for the preliminary definitions. We refer the reader to [171, 152, 178] for details on the UCLID5 syntax.

```
1 /* (Uclid5) Preliminary definitions */
2 const START : integer = 0; // The start index of the trace
```

```

3 const END : integer = 13; // The end index of the trace (ie, CT bound)
4 const END_TRACE : integer = 19; // The end index of the trace with padding
5 type state_t = {rxn_t, tag_t, val_t, trig_t, sch_t, pld_t}; // A state is a 6-tuple.
6 type path_t = [integer]state_t; // A path is a sequence of states indexed by integers.

```

The “Reaction Axiom Generator” generates UCLID5 axioms from the C AST in the If Normal Form. Again, we use the two reaction bodies in **ADASProcessor** as examples:

```

1 /* (Uclid5) Axioms of reaction 1 of ADASProcessor */
2 axiom(finite_forall (i : integer) in indices :: (i > START && i <= END) ==> (
3   (ADASModel_p_reaction_0(rxn(i))) ==> (((true ==> (
4     ((finite_exists (j : integer) in indices :: (j > i && j <= END_TRACE) && (
5       ADASModel_p_a_is_present(t(j))
6       && tag_same(g(j), tag_schedule(g(i), (500000000+0)))
7       && ADASModel_p_a(s(j)) == 0
8     )) // Closes finite_exists
9     && ADASModel_p_a_scheduled(d(i))))))
10    && (true ==> ((ADASModel_p_requestStop(s(i)) == 1)
11  )))))));
12
13 /* (Uclid5) Axioms of reaction 2 of ADASProcessor */
14 axiom(finite_forall (i : integer) in indices :: (i > START && i <= END) ==> (
15   (ADASModel_p_reaction_1(rxn(i))) ==> (((
16     (ADASModel_p_requestStop(s(i)) == 1) ==> (((ADASModel_p_out1(s(i)) == 1) && (
17       ADASModel_p_out1_is_present(t(i))))))
18  )))))));

```

As mentioned in Sec. 4.3, the generated reaction axioms are derived from a C AST in the If Normal Form and work in the same way as the other axioms by specifying relationships between variables across multiple time steps. For example, line 16 of the listing above is a direct translation of line 5 (*i.e.*, if (self->requestStop==1) lf_set(out1, 1);) in the C code listing above. The if statement becomes an implication where the if condition is the antecedent and the then branch of the if statement becomes the consequent.

The “MTL2FOL Generator” generates a First-Order Logic (FOL) property over the path in UCLID5 from the given MTL property. For the property in the running example, the generator produces the following:

```

1 /* (Uclid5) The FOL property translated from user-defined MTL property */
2 define p(i : step_t) : boolean = (((finite_forall (j0 : integer) in indices ::
3   (j0 >= i && j0 <= END && !isNULL(j0) && (
4     (pil(g(j0)) >= (pil(g(i)) + 0)) && (pil(g(j0)) <= (pil(g(i)) + 100000000))
5   )) ==> ((((((ADASModel_l_reaction_0(rxn(j0)))&&(((finite_exists (j1 : integer) in
6     indices
7     :: j1 >= j0 && j1 <= END && !isNULL(j1) && (((ADASModel_p_requestStop(s(j1)) == 1)
8     ))&&(
9     tag_same(g(j1), tag_delay(g(j0), 0))
10  )))))))) ==> ((((((finite_exists (j1 : integer) in indices :: j1 >= j0 && j1 <= END
11    && !isNULL(j1) && (((ADASModel_b_brakesApplied(s(j1)) == 1))) && (
12    (pil(g(j1)) >= (pil(g(j0)) + 0)) && (pil(g(j1)) <= (pil(g(j0)) + 550000000))
13  ))))))))));
14
15 /* (Uclid5) BMC: given an initial condition, the property needs to hold at step 0. */
16 property bmc_responsive : initial_condition() ==> p(0);

```

The “MTL2FOL Generator” recursively traverses the subformulas of the given MTL formula and builds up an FOL formula in a bottom-up direction. The generated FOL formula

implements the semantics of MTL shown in Sec. 4.2.

Checking and reporting results. Once the model is generated, it is passed into UCLID5, which generates an SMT file and invokes the Z3 SMT solver [43]. If the result is **UNSAT**, then the temporal properties are valid w.r.t. the formal model of the system. If the result is **SAT**, a counterexample trace is returned by UCLID5, which can help guide the user to revise the LF program. The process iterates until the unsafe behaviors are eliminated from the program. For the property in the running example, the LF VERIFIER reports **UNSAT** with no counterexamples found; therefore, the program satisfies the MTL property.

There are a few locations in the program where bugs can be introduced to invalidate the property.

Incorrect functional behavior.

The property could be invalidated if reactions have incorrect functional behavior, *e.g.*, the second reaction of **ADASProcessor** (lines 50-53) produces an output to the wrong port.

```
48 reaction(a) -> out1, out2 {=
49   if (self->requestStop==1)
50     lf_set(out2, 1); // Original: lf_set(out1, 1);
51 =} deadline(20ms) {=...=}
```

Incorrect composition.

The property is also invalidated when the components are not correctly composed, *e.g.*, when **LiDAR** is not connected to **ADASProcessor**.

```
73 // l.out -> v.in1           // Originally connected
74 c.out -> v.in2
75 v.out2 -> d.in
76 v.out1 -> b.inADAS after 5ms
77 p.out -> b.inPedal
```

Incorrect timing.

Timing behavior also plays a crucial role in determining the truth value of the property. If the timer in **LiDAR** has a large enough offset, the logical action in **ADASProcessor** has a larger delay, or the connection between **ADASProcessor** and **Brakes** has a larger delay, it could invalidate the given property.

```
5 timer t(11ms,17ms)           // Original: timer t(0,17ms)
40 logical action a(51ms)       // Original: logical action a(50ms)
76 v.out1 -> b.inADAS after 6ms // Original: v.out1 -> b.inADAS after 5ms
```

If any of these bugs (or any combination of them) occurs, the LF VERIFIER reports **SAT** and displays a counterexample path showing that the reaction in **Brakes** does not get triggered within 13 steps of the generated model. The user can then work through the counterexample to identify the specific bugs in the program.

Evaluation

Benchmarks. We evaluate our approach against a set of 22 LF programs drawn from real-life applications [176, 196, 123, 7, 56, 116, 65, 37], most of which are CPS applications with real-time properties, as well as benchmarks for Erlang [47], Lustre [65, 88, 36], actors [76] and RTOSes [50]. Thirteen of the 22 are safe examples and 9 are unsafe examples. The benchmark focuses on verifying the three types of properties introduced in Sec. 4.3: State properties are named Type I, dataflow properties Type II, and mixed properties Type III. The benchmark suite is also open-source on GitHub ².

Results. Table 4.1 shows the results of evaluating our approach against the verification benchmark suite. The experiments are conducted on a personal laptop running macOS version 11.7 with a 2.3 GHz 8-Core Intel Core i9 and 16GB RAM. The tool versions used are UCLID5 0.9.5 and Z3 4.8.8.

The LF VERIFIER correctly checks 21 out of 22 programs automatically. The one program the LF VERIFIER fails to check is **TrafficLight** because the C implementation used in one of the reaction bodies exceeds the subset of C currently supported by the LF VERIFIER at the time of writing. The results show that our tool is sound but incomplete. We expect a future version of LF VERIFIER supporting a larger subset of C will be able to handle this program. We discuss our plan to achieve this in Sec. 4.6.

We find that the majority of the computation time is spent on model generation: the median generation time for the UCLID5 model is 2.93 seconds, and the median generation time for the SMT formulas from the UCLID5 model is 12.15 seconds. However, once the SMT models are generated, solving them is very fast: the median solve time by Z3 is 0.31 seconds. The median total time is 15.9 seconds. Fig. 4.6 shows the distributions of our benchmark results. The total time scales almost linearly as the LF lines of code (LF LOC) and $\hat{\mathcal{CT}}$ increase.

4.6 Future Work

For future work on the LF VERIFIER, it is our plan to include larger programs (in terms of lines of LF code) in the benchmark suite and make the LF VERIFIER more scalable. To tame the explosion of complexity, we plan to explore two strategies: (i) detecting reactors in an LF program that do not affect the truth value of a given property using static analysis

²GitHub link: <https://github.com/lf-lang/lf-verifier-benchmarks>

Table 4.1: Benchmark results.

Program					LOC		TIME (seconds)			
	Type	#Rxns	$\hat{\mathcal{CT}}$	Safe?	LF	UCLID5	GenUclid	GenSMT	Z3	Total
ADASModel [196]	III	6	13	yes	78	875	3.35	17.72	1.72	22.79
AircraftDoor [123]	III	3	3	yes	37	402	3.59	2.98	0.02	6.59
Alarm [65]	III	2	2	no	24	338	2.93	2.43	0.01	5.37
CoopSchedule [50]	I	2	18	no	29	659	2.75	18.35	0.31	21.41
Elevator [36]	III	13	33	yes	118	1899	3.09	322.05	1038.02	1363.16
Election [37]	I	9	18	yes	73	1045	3.47	38.18	3.27	44.92
Election2 [37]	I	6	9	no	61	700	3.55	8.66	0.17	12.38
Factorial [36]	III	3	33	yes	38	461	3.73	17.73	13.15	34.61
Fibonacci [36]	III	5	55	yes	48	649	2.81	92.15	153.49	248.45
PingPong [76]	I	4	16	no	39	510	3.69	9.73	0.14	13.56
Pipe [47, 88]	I	5	24	no	54	2059	2.89	139.52	12.15	154.56
ProcessMsg [50]	I	4	19	yes	33	487	3.75	9.48	0.74	13.97
ProcessSync [50]	I	1	3	yes	12	284	2.78	2.21	0.01	5.00
Railroad [7]	I	10	20	yes	115	2333	2.93	155.21	4.33	162.47
Ring [47]	II	4	8	yes	39	897	3.68	12.15	0.07	15.9
RoadsideUnit [56]	I	8	21	yes	70	1536	3.78	70.29	0.62	74.69
SafeSend [47]	II	4	4	yes	37	479	2.70	3.78	0.03	6.51
Subway [65]	II	10	13	no	70	1097	2.85	27.11	0.31	30.27
Thermostat [116]	I	6	10	yes	59	652	2.80	10.61	0.14	13.55
TrafficLight [†] [36]	I	4	14	-	69	-	-	-	-	-
TrainDoor [176]	II	3	3	no	31	398	2.62	3.03	0.01	5.66
UnsafeSend [47]	II	4	4	no	38	478	2.84	3.66	0.02	6.52

[†] No result is produced because the C code in this program exceeds the analyzable subset.

and removing them from the axiomatic model, and (ii) using assume-guarantee reasoning to support compositional verification based on the modular structure of LF. In addition, we plan to support more complex C reaction bodies by expanding our “C AST Builder” or by offloading C code to external frameworks such as CBMC [93]. We also aim to support unbounded verification by potentially leveraging temporal induction [11]. Furthermore, we hope to support other LF features including physical actions, deadlines, modal models, etc.

A current limitation of the LF VERIFIER is that its axiomatic translation of reaction bodies only supports a restricted subset of C: statements that cannot be easily rewritten as logical axioms are conservatively over-approximated, and other target languages (C++, Rust, Python, TypeScript) are not supported at all. This limits the LF VERIFIER’s applicability to real-world LF programs whose reactions are often written in multiple languages and make use of idiomatic target-language features. A promising direction to address both

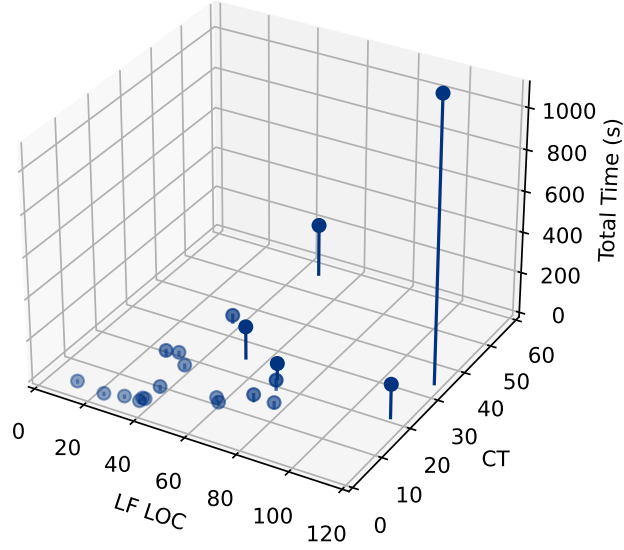


Figure 4.6: Distribution of benchmark results.

limitations is our follow-up work, POLYVER [38], a compositional verifier for polyglot systems that leverages Large Language Models (LLMs) together with language-specific verifiers (*e.g.*, CBMC for C, Kani for Rust) to synthesize pre-/post-condition contracts for reaction bodies under a counterexample-guided inductive synthesis (CEGIS) loop. Each reaction is abstracted by its contract, which is then plugged into an *operational* encoding of the system-level reactor behavior. This is a natural complement to the axiomatic encoding developed in this chapter. Integrating POLYVER’s contract-synthesis workflow with the LF VERIFIER would allow the benefits of both approaches (semantics-level soundness from axiomatic encoding, polyglot scalability from contract abstraction) to be combined, yielding an AI-in-the-loop verifier that handles the full diversity of reaction bodies found in practical LF programs.

Chapter 5

Quasi-Static Scheduling for Deterministic Timed Concurrent Models

“Simplicity is prerequisite for reliability.”

—Edsger W. Dijkstra, “How do we tell truths that might hurt?” 1975.

Modern cyber-physical systems (CPSs) aim to strike a balance between expressiveness, performance, and reliability. As use cases and operating environments become increasingly complex—for example, driving an autonomous vehicle in a densely populated city or flying the Ingenuity Mars Helicopter autonomously over complex Martian terrain—CPS designers aim to give systems expressive behavior to handle such complexity. However, certain use cases that demand rich behavior pose stringent performance requirements.

Despite steady progress, using multi-core architectures in CPSs remains challenging. For example, predicting timing properties on these architectures, such as a sequential program’s worst-case execution time (WCET), is still an active research area [133]. Furthermore, concurrent programming is known to be notoriously difficult, especially using threads [101]—still a mainstream programming model today. To balance the trade-offs between expressiveness, performance, and reliability, researchers have extensively studied concurrent models of computation (MoCs) and their formal properties. In particular, models that can either fully or partially derive schedules at compile time receive continued attention, due to their attractive properties such as compile-time schedule validation and low runtime overhead. The technique of making most scheduling decisions at compile time and only the necessary decisions at runtime is known as *quasi-static scheduling* [105]. Prominent models amenable to deriving schedules at compile time include synchronous dataflow (SDF) [106], Boolean-controlled dataflow (BDF) [29], scenario-aware dataflow (SADF) [177], and logical execution time (LET) [85], just to name a few. In this chapter, we treat fully static scheduling of some models as a subset of quasi-static scheduling.

Challenges. For the aforementioned models, existing solutions for quasi-static scheduling show commonalities. However, these techniques appear isolated, and no prior work has

looked at them through a unified lens. This fragmentation makes it difficult to understand their subtle differences, complicating systematic comparison and evaluation. When new MoCs emerge, it is unclear whether existing proven techniques can be readily applied. Additionally, fragmented methods also lead to repeated, isolated implementations, hindering code reuse.

Our Solution. In this chapter, we present the first *unified* approach to quasi-static scheduling for concurrent MoCs. Our approach generalizes lessons learned from SDF, BDF, SADF, and LET into a common scheduling framework based on a new intermediate formalism called state space finite automata (SSFA). This unified SSFA-based approach provides a frame of reference for comparing existing quasi-static scheduling techniques and helps identify a class of models to which the approach is applicable. We use the reactor model [127] as a concrete use case of our proposed scheduling approach and demonstrate an implementation in the reactor model’s code generator—the LINGUA FRANCA [129] compiler.

Concretely, this chapter presents the following key contributions:

1. We present the first unified approach to quasi-static scheduling based on an intermediate formalism called state space finite automata (SSFA) (Sec. 5.3), which bridges the gap between high-level MoCs and low-level executable schedules (Sec. 5.1).
2. We further define deterministic, timed, concurrent models (DTCMs), a class of MoCs to which SSFA-based scheduling applies. We discuss key properties of our proposed technique, including anomaly-free timing (without requiring WCETs) and analyzable worst-case makespan (Sec. 5.3).
3. We concretely demonstrate how our unified SSFA-based scheduling approach applies to the reactor model and integrate it directly into its existing toolchain—the LINGUA FRANCA (LF) compiler (Sec. 5.4).
4. We evaluate the effectiveness of our approach by measuring the performance of quasi-statically scheduled LF programs against that of the default, dynamically scheduled LF programs using a set of benchmarks written in a synchronous subset of LF. Our results demonstrate that quasi-statically scheduled LF programs reduce runtime overhead, achieving an average improvement of $9\times$ (Sec. 5.5). In addition, we evaluate SSFA’s analyzable worst-case makespan through a case study on a satellite attitude controller (Sec. 5.5).

5.1 Background and Related Work

Concurrent Models of Computation

A model of computation (MoC) is a mathematical abstraction of a computing device [78]. An MoC provides a formal framework either for designing or for reasoning about a computing

device. For example, an MoC, such as Communicating Sequential Processes [73], can be used to design a signal processing algorithm, which together with a compiler and a runtime can result in an executable. However, the same MoC can also be used to model the set of all possible behaviors of an algorithm and to formally verify statements about the algorithm. In this chapter, we are concerned with the former, i.e. using MoC as a mechanism to design a computation, and in particular we present a methodology for producing quasi-static schedules from various MoCs.

MoCs are distinguished by their mechanisms for modeling concurrency and time. Some MoCs, such as the Finite State Machine and the Turing Machine, do not include primitives for modeling concurrent processes and are known as Sequential MoCs. Concurrent MoCs are increasingly popular because the end of Dennard scaling has spawned the age of multi-cores, and recently, even low-power microcontrollers, such as the ESP32, have adopted multi-core architectures. Concurrent MoCs are naturally better suited for mapping onto parallel hardware.

In this chapter we will discuss the generation of executable schedules from various forms of Dataflow Process Networks [154], where computation is expressed as a set of connected actors with firing rules; Logical Execution Time [85], a task model with precedence constraints among tasks and where each task has a logical execution time; and reactors [127], where computation is expressed as a set of event-triggered actors with discrete-event semantics.

Quasi-Static Scheduling for Concurrent MoCs

In the literature of concurrent MoCs, the ability to derive schedules at compile-time from MoC semantics has long been a key topic of interest. Lee and Messerschmitt [106] propose the synchronous dataflow (SDF) model, where each concurrent actor specifies the number of tokens received from a channel to trigger a firing, as well as the number of tokens sent into a channel after a firing. By explicitly giving token production and consumption rates, SDF enables efficient static scheduling, which begins by solving a balance equation representing token distributions in the system [115]. Intuitively, each iteration of the static schedule must return the distribution of tokens to the starting state, effectively completing a hyperperiod.

SDF enables fully static compile-time scheduling, at the cost of rigidity in its behavior. To enable more dynamicity without entirely sacrificing analyzability, Buck and Lee [29] introduce the Boolean-controlled dataflow (BDF) model, which provides **SWITCH** and **SELECT** actors that accept Boolean tokens that route token flow and modulate token production rate at runtime. Given the dynamic nature of BDF, only certain BDF systems can yield compile-time schedules, subject to constraints such as observability of the Boolean token stream that controls token routing [28]. A hyperperiod is similarly obtained from solving a (now symbolic) balance equation.

Another MoC that extends SDF with dynamism is scenario-aware dataflow (SADF) [58]. SADF models a system using multiple SDF graphs with nondeterministic transitions between them at the end of an SDF graph's iteration. To implement SADF, Kampenhout et al. [83] borrow the **SWITCH** and **SELECT** actors to perform scenario change and propose using a rolling

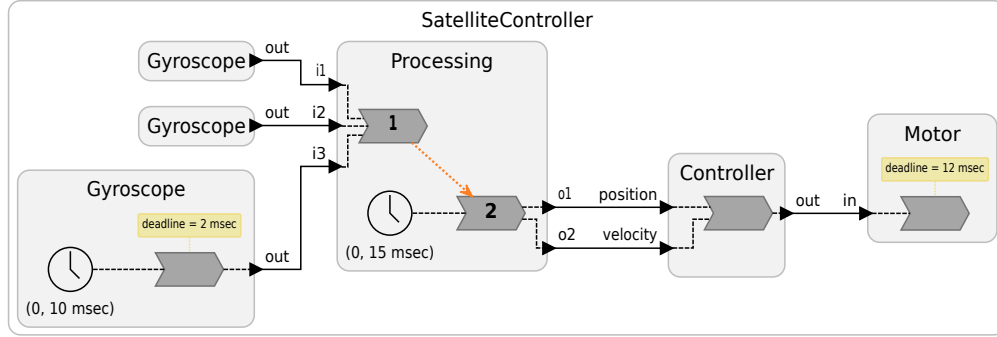


Figure 5.1: A satellite attitude control system.

static-order schedule, which at runtime concatenates the static schedule of the next scenario (which itself is an SDF graph) to a rolling schedule as soon as the next scenario is determined. This approach exemplifies quasi-static scheduling since each scenario’s schedule is determined at compile-time, while the switching between scenarios is determined at runtime.

Dataflow models are typically suitable for streaming applications such as digital signal processing. For cyber-physical systems, which benefit from the ease of specifying precise timing behavior, the Logical Execution Time (LET) model [85, 59] has gained traction. The LET model defines deterministic system behavior by specifying that tasks appear to read inputs instantaneously at one time point, compute for a fixed amount of time without interacting with other concurrent tasks, then instantaneously produce outputs at a later point in time. LET is known to be amenable to static scheduling [70] based on the hyperperiod of the LET tasks [52] and precedence constraints [74].

While there has been work on surveying and comparing MoCs—see, e.g., a recent survey from Roumage et al. [160]—to the best of our knowledge, our work provides the first unified framework for quasi-static scheduling.

5.2 Running Example

Fig. 5.1 shows the block diagram of a satellite attitude control system. The three **Gyroscope** actors each sample a physical, or simulated, gyroscope sensor and are triggered every ten milliseconds. The **Processing** actor combines a sample from each of the three gyroscopes into an average. The **Processing** actor also produces a position and velocity estimate every 15 milliseconds, which is fed into the **Controller** actor implementing a PID controller. Finally, the control output is forwarded to the **Motor** actor, controlling a physical (or simulated) stepper motor attached to a reaction wheel controlling rotation along one axis.

Attitude control of a satellite is clearly a real-time problem and the timeliness of the interaction between software and hardware is paramount. The program has two annotated deadlines. First, the **Gyroscope** actors have a 2-millisecond deadline on the reaction that

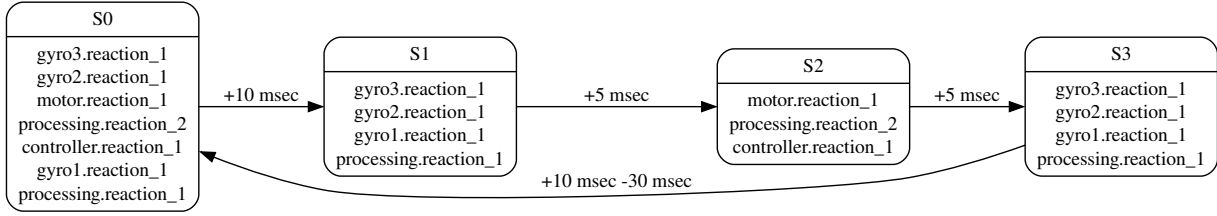


Figure 5.2: Hyperperiod of the satellite attitude controller in Fig. 5.1.



Figure 5.3: General quasi-static scheduling workflow based on SSFA. The colors, red and blue, refer to two main stages in the workflow: SSFA generation (Sec. 5.3) and DAG scheduling (Sec. 5.3).

samples the sensors. This means that the code sampling the sensors should finish executing within two milliseconds from the time that the timer triggers. Second, the **Motor** actor has a 12-millisecond completion deadline on the reaction driving the stepper motor, meaning that this reaction must be completed within 12 milliseconds after the triggering of the 15-millisecond timer in the **Processing** actor. This periodic program can conveniently be expressed in a DTCM, such as SDF, BDF, SADF, or LET.

5.3 Quasi-Static Scheduling based on SSFA

In Sec. 5.1, prior quasi-static scheduling work reveals an important pattern: generating schedules at compile-time typically requires finding the system’s hyperperiod, or multiple hyperperiods if the system has dynamic behaviors (e.g., BDF or SADF). A hyperperiod encodes a sequence of system state transitions in which the system returns to its starting state at the completion of the sequence. For example, Fig. 5.2 shows the hyperperiod of the running example. Here, a state is defined by an ID and a set of task invocations, and a state transition is represented by time advances, a representation common for time-triggered models like LET. We note that if the running example is explicitly specified in a dataflow model, *e.g.* SDF, a hyperperiod similar to Fig. 5.2 can still be identified by techniques such as Ghamarian et al. [61].

Fig. 5.3 shows our generalized quasi-static scheduling methodology. The input to the methodology is a system specified in a concurrent MoC, such as the MoCs mentioned in Sec. 5.1. From the system specification, a state space finite automaton (SSFA) is constructed from the semantics of the MoC. Intuitively, an SSFA is an FSM in which each node has an associated directed acyclic graph (DAG) representing task executions in a hyperperiod. The

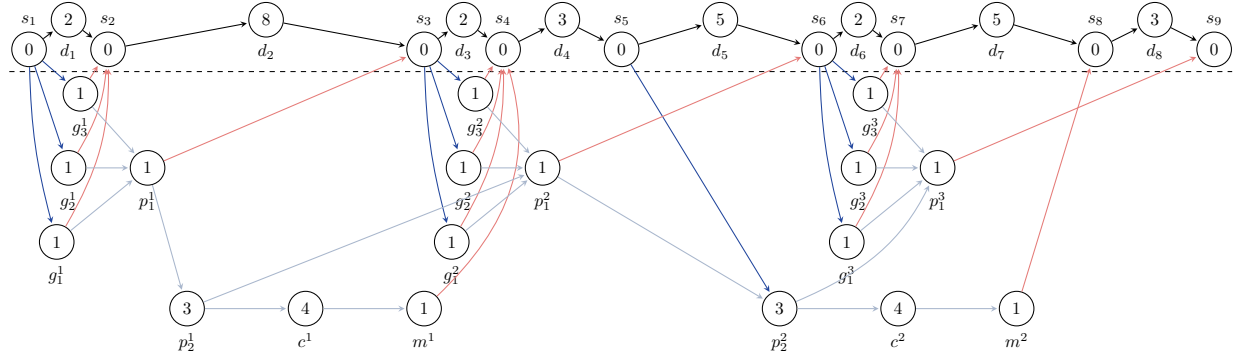


Figure 5.4: DAG representation after conversion from the hyperperiod in Fig. 5.2, showing release edges in blue, deadline edges in red, and data dependency edges in gray, with the virtual path above the dashed line. The node names are the initials of the component names. The numbers in the nodes are the WCET bounds of each task, and the nodes above the dashed line are the nodes on the virtual path. Sync nodes are identified by s , marking the release and deadline of tasks; dummy nodes are identified by d , marking time intervals between sync nodes. Details of virtual nodes are introduced in Sec. 5.3.

FSM further specifies guarded transitions that can be taken at the end of a DAG's execution.

In this section, we define DAGs and SSFAs, and discuss how to generate both of them from a class of MoCs called deterministic timed concurrent models (DTCMs).

Directed Acyclic Graphs (DAGs)

DAGs are often used in real-time applications, such as automotive and avionics, to model real-time computing tasks and their precedence constraints. This subsection introduces basic concepts and notations for real-time DAG task modeling.

Example. Fig. 5.4 shows the DAG generated from the system hyperperiod in Fig. 5.2.

Task Model. A DAG task set is characterized by $(\mathcal{G}, P) = DAG \in DAGs$, in which $\mathcal{G}$ is a graph that defines the set of tasks and their precedences, and P denotes the period of the DAG task set. The graph $\mathcal{G}$ consists of $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i\}$ is a set of n nodes representing n tasks, and $\mathcal{E} = \{e_{ij}\}$ is a set of directed edges representing the precedence relation between the tasks. For any two nodes v_i and v_j connected by a directed edge e_{ij} , v_j can start execution only if v_i has finished its execution. Given the edge e_{ij} , v_i is the predecessor of v_j . Each task v_i is a non-preemptive sequential computing workload, with its WCET denoted as C_i .

Node-level Timing Attributes. The timing constraints of each node can be defined through four attributes: earliest starting time (EST), earliest finishing time (EFT), latest starting time (LST), and latest finishing time (LFT). The EST is the earliest time a node

can start executing, equaling the maximum of its predecessors' EFTs. Similarly, the LFT defines the latest time a node can finish its execution to meet the deadline, equal to the minimum of its successors' LSTs. Note that EST, EFT, LST, and LFT are defined based on the precedence relations defined by $\mathcal{E}$ by *assuming that the execution time of a task is equal to its WCET*. If v_i has no predecessor, then $t_i^{\text{EST}} = 0$. If v_i has no successor, then $t_i^{\text{LFT}} = P$. Furthermore, we have

$$\begin{cases} t_i^{\text{EST}} = \max\{t_j^{\text{EFT}} \mid e_{ji} \in \mathcal{E}\}, \\ t_i^{\text{LFT}} = \min\{t_j^{\text{LST}} \mid e_{ij} \in \mathcal{E}\}, \\ t_i^{\text{EFT}} = t_i^{\text{EST}} + C_i, \\ t_i^{\text{LST}} = t_i^{\text{LFT}} - C_i. \end{cases} \quad (5.1)$$

These timing attributes can be computed iteratively using a fixed-point approach, following the classic ASAP (as soon as possible) and ALAP (as late as possible) analysis [141]. Specifically, all nodes are initially assigned $t_i^{\text{EST}} = 0$ and $t_i^{\text{LFT}} = P$, where P is the period of the DAG task set. At each iteration, values are updated according to the equations above until convergence, i.e., when no values change between successive iterations. Since the critical path of the DAG contains at most n nodes, convergence is guaranteed within n iterations [186]. The overall time complexity of this computation is $\mathcal{O}(n^3)$.

Virtual Path. When mapping a multi-rate task set to a single-rate DAG, job-level timing constraints must be expressed. For example, the second reaction of gyroscope 1, g_1^2 , should only be able to start in the second gyroscope period, i.e., after 10 ms. Additionally, that gyroscope reaction has a deadline of 2 ms from its start, i.e., at 12 ms. To enforce these timing constraints, we borrow the concept of *virtual nodes* from Verucchi et al. [186]. There are two types of virtual nodes, *dummy nodes* with a predefined execution time C_{d_j} for dummy node d_j and *sync nodes* with zero execution time. A *virtual path* (the top row in Fig. 5.4) can be constructed using the virtual nodes, alternating S sync and $S - 1$ dummy nodes. The total execution time of the virtual path is equal to the period of the DAG task set, i.e., $\sum_{j=1}^{S-1} C_{d_j} = P$. Through the defined execution time of the dummy nodes, it follows that for each sync node s_l ,

$$t_{s_l}^{\text{EST}} = t_{s_l}^{\text{EFT}} = t_{s_l}^{\text{LST}} = t_{s_l}^{\text{LFT}} = \sum_{i=1}^{l-1} C_{d_i} = P - \sum_{j=l}^{S-1} C_{d_j}. \quad (5.2)$$

The nodes in the virtual path can be designed such that all unique offset and deadline values are represented by a sync node. Adding an edge from the appropriate sync node s_l to a task v_k , or vice versa, respectively, results in the relation

$$O_k = t_{s_l}^{\text{EFT}} \leq t_k^{\text{EST}} \quad \text{or} \quad t_k^{\text{LFT}} \leq t_{s_l}^{\text{LST}} = D_k, \quad (5.3)$$

which enforces the timing constraints within the DAG task set.

For the offset of the second reaction of gyroscope 1, g_1^2 , the sync node s_3 is created, which is preceded by two dummy nodes with a total execution time of 10 ms. By adding

a precedence constraint from s_3 to g_1^2 , that gyroscope reaction cannot start before 10 ms, enforcing its offset. Similarly, for its deadline constraint, sync node s_4 is created, which is succeeded by multiple dummy nodes with a total execution time of 18 ms. By adding a precedence constraint from g_1^2 to s_4 , the gyroscope reaction must finish no later than 18 ms before the hyperperiod, i.e., before 12 ms.

Trivial Schedulability. A DAG task $(\mathcal{G}, P)$ is defined to be *trivially schedulable* on M processors, if its makespan is shorter or equal to its period, and if its width is less or equal to the number of available processors [179]. Sun et al. [179] proved that a trivially schedulable DAG task is schedulable under any work-conserving executor (Lemma IV.3 and Lemma IV.4 in [179]). Additionally, they showed that any schedulable DAG task can be transformed into a trivially schedulable DAG task by adding edges to it. This transformation leads to the formulation of the *Edge Generation Scheduler* (EGS), in which edges are added iteratively to a DAG until it is trivially schedulable.

State Space Finite Automata (SSFA)

An SSFA is an extended finite state machine designed specifically for representing the behavior of concurrent MoCs. Formally, an SSFA is defined as a tuple:

$$SSFA = (M, V, D, T, G, m_0),$$

where:

- M is a finite set of modes,
- V is a finite set of variables, with cross product of their domains represented as $\mathcal{V} = \prod_{v \in V} \text{domain}(v)$,
- $D : M \rightarrow \text{DAGs}$ is a function assigning a directed acyclic graph (DAG) to each mode $m \in M$,
- $T \subseteq M \times M$ is a set of transitions between modes,
- $G : T \times \mathcal{V} \rightarrow \mathbb{B}$ is a guard function mapping the transition and variables to a Boolean condition,
- $m_0 \in M$ is the initial mode.

The execution semantics of an SSFA are as follows:

1. The SSFA begins execution in the initial mode m_0 .
2. When the SSFA enters a mode $m \in M$, it executes the DAG specified by $D(m)$ to completion.

3. Upon completion of the DAG execution, the guard conditions associated with the outgoing transitions of the current mode are evaluated using G .
4. If a guard condition evaluates to true, the corresponding transition is taken, and the SSFA moves to the next mode. If several guards are true simultaneously, one of the enabled transitions between modes is taken nondeterministically.

The behavior of the running example can be represented by a trivial SSFA with a single mode containing the DAG shown in Fig. 5.4 with a transition back to itself that has a default guard *true*. In Sec. 5.3, we show how the SSFA for dynamic models like BDF and SADF can have more modes with nontrivial guards. We further show a nontrivial SSFA in Fig. 5.10 in Sec. 5.4.

Mapping High-Level MoCs to SSFA

Existing Models to SSFA. We now discuss how to map the MoCs surveyed in Sec. 5.1 to SSFA.

SDF Since SDF defines a unique recurring behavior, the resulting SSFA therefore has only a single mode with a trivial self-transition. The DAG within the SSFA mode can be obtained by running the algorithm in Appendix I of Sih [172].

BDF Buck [28] notes that generating compile-time schedules requires knowledge of exact values of the emitted Boolean tokens, and an “acyclic precedence graph” (DAG) can be constructed for each case of Boolean token values. To generate a corresponding SSFA, each possible case of Boolean token values produces an SSFA mode, with the corresponding DAG constructed from a resulting SDF graph after filling in the concrete case of Boolean values. The modes have all-to-all transitions so that any mode can be switched to at runtime based on the upcoming Boolean tokens.

SADF Each scenario becomes an SSFA mode. The SDF graph within each scenario is similarly transformed into a DAG based on Sih [172]. Transitions between SADF scenarios become SSFA mode transitions, and transition guards in SSFA are derived from the SADF model.

LET The LET tasks are by design periodic, and the schedule repeats after each iteration of the hyperperiod of all task periods. This behavior results in a unique SSFA mode. A DAG can be constructed from the time and data dependencies specified among the LET tasks.

From lessons learned from existing MoCs, we summarize a list of observations these MoCs satisfy in order to map to SSFA.

1. The MoC defines a finite set of execution phases.

2. A phase often contains a hyperperiod, *i.e.* a unique partial order of task firings that eventually returns the system state to a starting state.
3. The MoC defines transitions, either deterministic or nondeterministic, between phases.

SDF and LET define a unique execution mode given the static nature of their task firings, while SADF defines a mode for each scenario. BDF defines a mode for a unique value set of its Boolean control tokens. The execution within a mode is fully deterministic, while mode switching depends on runtime conditions and can be nondeterministic, as SADF shows.

Deterministic Timed Concurrent Models (DTCMs). From the above requirements, we identify a class of MoCs, named Deterministic Timed Concurrent Models (DTCMs), where the proposed SSFA-based scheduling approach applies. A DTCM's execution can be represented as a finite set of *phases* P . Each phase $p \in P$ is a finite sequence of states $(s_i, s_{i+1}, \dots, s_{i+n}) \in S^n$. The exact definition of a *state* $s \in S$ should be provided by the DTCM, and S denotes the set of all possible system states defined by the DTCM. Within each phase p , the evolution of system states is described by a *deterministic* state-transition relation $T_p \subseteq S \times S$, where each transition $(s_i, s_j) \in T_p$ takes the system from a state s_i to a unique successor state s_j . For example, SDF typically defines the state to be a distribution of tokens in the channels. In Sec. 5.4, we show another example of state definition for the reactor model. Transitions between phases are also defined by the DTCM, and they can be nondeterministic. In SADF, nondeterministic transitions between scenarios exemplify nondeterministic phase transitions [58].

Although there are no general rules for identifying phases for a DTCM's execution, a classical approach involves explicit state space exploration [61, 29, 122] to identify *hyperperiods*. A hyperperiod h is a sequence of states $h = (s_i, s_{i+1}, \dots, s_{i+n})$ such that the next state s_{i+n+1} returns to s_i , assuming no phase transition occurs. To identify a hyperperiod, the DTCM defines a predicate $\mathcal{H} : S \times S \rightarrow \mathbb{B}$ that determines whether the second state has returned to the first state, completing a hyperperiod. Then it unrolls from a starting state s_0 until a minimal hyperperiod $h = (s_i, s_{i+1}, \dots, s_j)$ is found such that $\mathcal{H}(s_i, s_{j+1}) = \text{true}$ for some $j \geq i \geq 0$. This process yields a sequence of states forming a lasso structure. The leading transient states can thus be categorized into one phase $p_0 = (s_0, s_1, \dots, s_{i-1})$, and the remaining hyperperiod into another phase $p_1 = (s_i, s_{i+1}, \dots, s_j)$. For certain DTCMs, more efficient state space exploration techniques exist, such as solving SDF's balance equations [106]. In general, the DTCM should supply a technique to obtain the minimal hyperperiod based on its semantics.

Once the phase set P is obtained, the DTCM should provide a method to convert the task firings involved in each $p \in P$ to a DAG, as described earlier in Sec. 5.3. A specific example for the reactor model is provided in Sec. 5.4. Once a DAG for a phase p is constructed, a corresponding SSFA mode can be produced—this process is repeated for each phase $p \in P$. Lastly, transitions between SSFA modes can be added based on guarded transitions defined by the DTCM semantics.

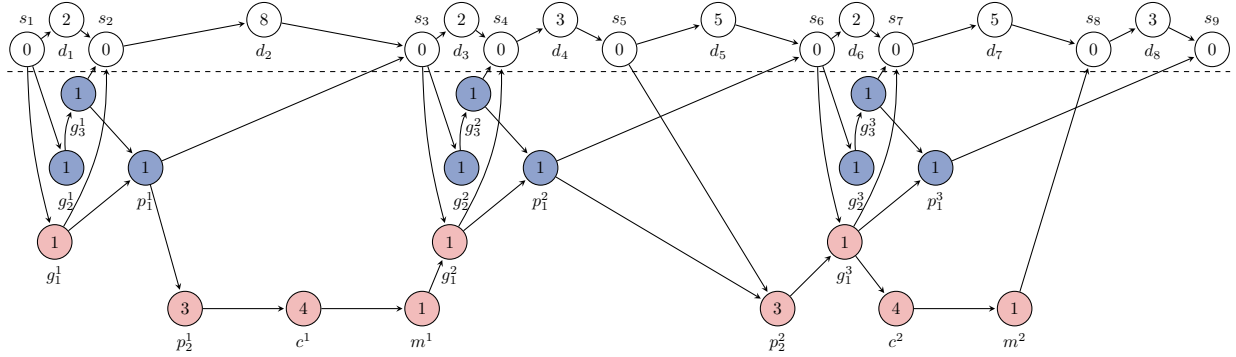


Figure 5.5: Sample output from a DAG scheduler, a transitive reduction of the DAG in Fig. 5.4 with width $w = 2 + 1$, with $+1$ for the virtual path. The colors of the nodes show a valid partitioning to 2 available cores.

Scheduling SSFA by Graph Partitioning

To schedule the SSFA, a partitioned DAG scheduler is invoked on each DAG contained in the SSFA states. The DAG scheduler takes an unpartitioned DAG as input, partitions the DAG based on the number of processor cores available, and outputs graph partitions, which form a spatial mapping between tasks and execution platforms with timing and inter-task dependencies. Given an unpartitioned DAG, a DAG scheduler can designate nodes to any partition it deems fit. The scheduler can modify edges if the output partitioned DAG satisfies all dependencies in the input unpartitioned DAG. In addition, the DAG scheduler must *sequentialize* each partition by adding edges such that there exists a path in the output DAG passing through all nodes within the same partition. The resulting DAGs partitioned by this type of DAG scheduler are *trivially schedulable*, as defined by Sun et al. [179] and discussed in Sec. 5.3. A trivially schedulable DAG can easily be partitioned using paths that cover the DAG, as detailed in Algorithm 1 of [179], with an example shown in Fig. 5.5.

We note that, in the real-time scheduling literature, there is a large body of prior work on partitioned scheduling [185]. In principle, any DAG partitioning algorithm could be plugged into this step, as long as there is post-processing that guarantees each partition is sequentialized. To execute a partitioned DAG, there is prior work as well, e.g., Huang et al. [75] and Henzinger et al. [69]. In our demonstration of the reactor model in Sec. 5.4, we use a strategy similar to Henzinger et al. [69].

After scheduling all DAGs in the SSFA, we have generated the quasi-static schedules for the DTCM. Compared to prior work’s static or quasi-static schedules (such as SDF’s acyclic precedence graphs [115] and LET’s table of possible release timing [68]), the SSFA formalism is operationally equivalent, in the sense that task dependencies and task-to-core mappings are preserved. However, the SSFA-based representation differs in structure and expressiveness: instead of representing a schedule as a flat precedence graph (as in SDF) or a time-triggered release table (as in LET), SSFA encodes scheduling decisions within

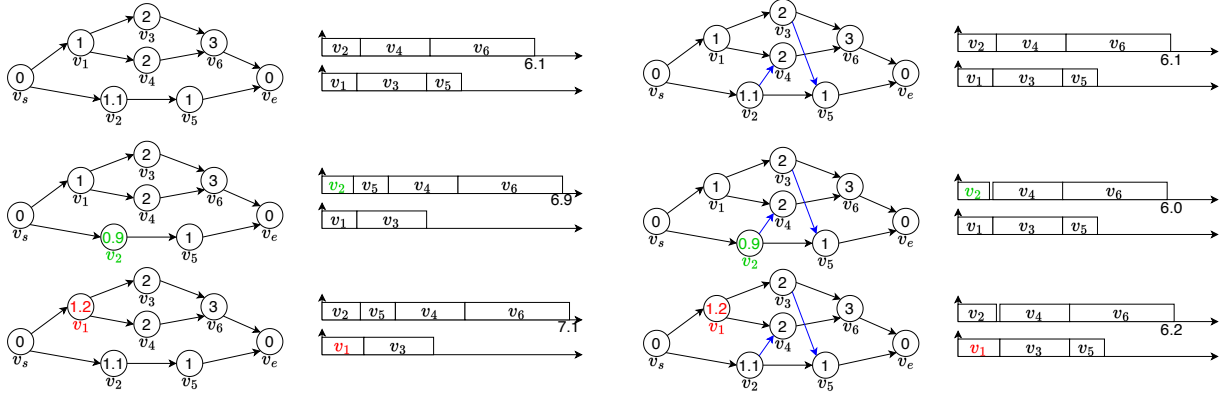


Figure 5.6: Scheduling anomaly example.

the modes of an extended finite-state machine, with a DAG in every mode and transitions guarded by logical conditions that define when the system evolves to the next mode.

Formal Properties

In this section, we discuss formal properties provided by our SSFA-based scheduling approach.

Anomaly-Free Timing. A timing anomaly in real-time systems refers to a situation where a local shorter execution time (like a cache hit) leads to an unexpected global effect of an increase in the overall execution time [131, 97]. Timing anomalies can occur with the dynamic scheduling of tasks (reactions) on multiple execution units. We now show that our proposed SSFA-based quasi-static scheduling guarantees that no timing anomalies will occur on multi-core processors. It is important to note that the monotonicity property holds *without* requiring knowledge of WCET bounds.

Example Fig. 5.6 shows two scenarios that can cause scheduling anomalies under work-conserving scheduling for nontrivially schedulable DAGs. The first row shows a DAG task and a corresponding schedule using a work-conserving scheduler. The DAG on the right is modified to be trivially schedulable by adding two edges, yielding the same schedule as on the left. In the second row, we consider a scenario where the execution time of v_2 is decreased by 0.2 ms, causing it to finish earlier than v_1 . As a result, v_5 can be started, delaying the execution of v_4 and v_6 , leading to a makespan increase of 0.8 ms. This scenario is the typical scheduling anomaly example where the decrease of an individual execution time leads to an increase in the overall makespan. On the right, in the trivially schedulable DAG, the execution order is not altered since v_5 needs to wait for v_3 to finish, and the processor stalls until v_4 is ready. Thus, the makespan does not increase but decreases by 0.1 ms instead.

The third row shows that the same reordering in the work-conserving schedule can be triggered by increasing an individual execution time, causing a larger increase in the overall makespan.

Formal definition Consider a DAG task set $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with estimated execution times $\mathbf{C} \in \mathbb{R}^{|\mathcal{V}|}$ associated with each node $v_i \in \mathcal{V}$ and a scheduling algorithm yielding an expected makespan M if all nodes execute with their respective C_i . Suppose a node executes with a different computation time $\bar{C}_i = C_i + \Delta_i$. A scheduling algorithm is *monotonic* if the makespan, denoted as M , resulting from the actual computation time satisfies $M(\bar{\mathbf{C}}) \leq M(\mathbf{C}) + \max\{\Delta_i, 0\}$.

Theorem 3 (Monotonicity of trivially schedulable DAG tasks). *When executing a trivially schedulable DAG task in a work-conserving manner, the monotonicity property is guaranteed.*

Proof. The makespan of a trivially schedulable DAG is equal to the length of its longest path, i.e., $M(\mathbf{C}) = \max_{p \in \mathcal{P}} L(p, \mathbf{C})$. The length of a path $p \in \mathcal{P}$ under execution times $\mathbf{C}$ is $L(p, \mathbf{C}) = \sum_{v_i \in p} C_i$. Under the new execution times $\bar{\mathbf{C}}$, the lengths of the paths in the DAG change as follows:

$$L(p, \bar{\mathbf{C}}) = \begin{cases} L(p, \mathbf{C}) + \Delta_i, & \text{if } v_i \in p, \\ L(p, \mathbf{C}), & \text{otherwise.} \end{cases} \quad (5.4)$$

Therefore, $M(\bar{\mathbf{C}}) \leq \max\{M(\mathbf{C}) + \Delta_i, M(\mathbf{C})\} = M(\mathbf{C}) + \max\{\Delta_i, 0\}$. ■

Corollary 3.1. *If multiple nodes, denoted as N , alter their execution times, the resulting makespan is bounded by $M(\bar{\mathbf{C}}) \leq M(\mathbf{C}) + \sum_{v_i \in N} \max\{\Delta_i, 0\}$.*

Corollary 3.2. *Since any partitioned schedule with predefined sequentialization on each processor can be expressed as a trivially schedulable DAG (see Theorem IV.5, necessity part in [179]), any work-conserving partitioned scheduling algorithm is monotonic.*

Monotonicity implies two properties: (i) If the execution time of the nodes is less than their estimated execution time, the makespan does not increase. This property is often called *sustainability* [31] and guarantees the absence of timing anomalies. (ii) If the nodes exceed their execution times by some delta, the increase of the overall makespan is bounded by the sum of the deltas.

Analyzable Worst-Case Makespan. A partitioned, trivially schedulable DAG produced by the proposed workflow yields a simple procedure to check the worst-case makespan up to some node or for the entire hyperperiod.

Definition 5 (Partial makespan up to node n). *The partial makespan up to a node n , denoted as $\bar{L}(n)$, is defined recursively as*

$$\bar{L}(n) = \begin{cases} wcet(n) & \text{if } U(n) = \emptyset, \\ \max_{u \in U(n)} L(u) + wcet(n) & \text{otherwise.} \end{cases}$$

where $wcet : \mathcal{V} \rightarrow \mathbb{T}$ returns the WCET of a DAG node within a time domain $\mathbb{T}$ and $U : \mathcal{V} \rightarrow \mathcal{P}(\mathcal{V})$ maps a DAG node n to a set of its immediately upstream DAG nodes.

Example 1. To validate the deadline of 12ms of the first invocation of the motor reaction in Fig. 5.1, we calculate the partial makespan up to m^1 in Fig. 5.5, i.e. $\bar{L}(m^1)$.

$$\begin{aligned}
\bar{L}(m^1) &= \bar{L}(c^1) + wcet(m^1) \\
&= \bar{L}(p_2^1) + wcet(c^1) + wcet(m^1) \\
&= \bar{L}(p_1^1) + wcet(p_2^1) + wcet(c^1) + wcet(m^1) \\
&= \max(\bar{L}(g_3^1), \bar{L}(g_1^1)) + wcet(p_1^1) + wcet(p_2^1) + wcet(c^1) + wcet(m^1) \\
&= \max(wcet(g_2^1) + wcet(g_3^1), wcet(g_1^1)) + wcet(p_1^1) + wcet(p_2^1) + wcet(c^1) + wcet(m^1) \\
&= \max(1 + 1, 1) + 1 + 3 + 4 + 1 = 11 \leq 12
\end{aligned}$$

Therefore, the deadline of 12ms of the first invocation of the motor reaction can be guaranteed to hold.

So far we have introduced a generic methodology to generate quasi-static schedules for a class of DTCMs and their formal properties. In the subsequent section, we demonstrate applying the methodology to a specific new DTCM—the reactor model—and show how to implement the methodology in concrete tooling.

5.4 Applying Methodology to the Reactor Model

In this section, we demonstrate an application of the general SSFA-based scheduling methodology to an emerging MoC—the reactor model [127], which has a front-end coordination language called LINGUA FRANCA (LF) [129]. The reactor model in LF decouples the passage of physical time from logical time, except when interactions with the physical world are involved (through timers, sensors, and actuators). In part, because LF allows sporadic events, the scheduling of LF programs is carried out dynamically at runtime. While the ideal, logical timing behavior is specified in the LF code, alignment with physical time is executed in a best-effort way, and there are no guarantees about the absence of timing anomalies on multicore systems or deadline compliance at the MoC level, since they are the concerns of LF’s underlying scheduling policies. The running example in Fig. 5.1 is an automatically generated diagram from the LF toolchain.

We now apply our SSFA-based quasi-static scheduling technique to a *synchronous* subset of LF, with its syntax shown in Fig. 5.7. We mainly exclude two LF language features: actions and physical connections, effectively only allowing timer-driven tasks and event-driven tasks from logical connections.

The user first provides an LF program with *optional* WCET annotations, each of which indicates the WCET of a reaction on a certain execution platform. WCETs, if specified, help guide the DAG schedulers to generate better schedules. In practice, they may not be known

```

    Model ::= TargetDecl Preamble * Reactor+
    TargetDecl ::= target id
    Reactor ::= reactor id? Params? { ReactorBody * }
    Params ::= ( Parameter ( , Parameter ) * )
    ReactorBody ::= Preamble | StateVar | Input | Output | Timer
                  | Reaction | Instantiation | Connection
    StateVar ::= state id ( : Type )? ( = Expression )?
    Timer ::= timer id ( Expression( , Expression )? )
    Connection ::= VarRef -> VarRef ( after Expression )?
    Instantiation ::= id = new id ( Parameter ( , Parameter ) * )
    Literal ::= true | false | INT | FLOAT
    Reaction ::= reaction ( TriggerList? ) ( -> EffectList )? Code

    Expression ::= Literal | Time | Code
    TriggerList ::= TriggerRef ( , TriggerRef ) *
    EffectList ::= VarRef ( , VarRef ) *
    Parameter ::= id : Type = Expression
    Input ::= input id ( : Type )?
    Output ::= output id ( : Type )?
    Type ::= time | id | Code
    TriggerRef ::= BuiltinTrigger | VarRef
    VarRef ::= id ( . id ) *
    Preamble ::= Code
    Code ::= { = id* = }

```

Figure 5.7: Syntax of the synchronous subset of LF.

or the estimates may be inaccurate or pessimistic. Our procedure will still generate correct schedules, though timing optimality cannot be assured. Moreover, as we show in Sec. 5.3, the schedules are free from timing anomalies, in that if an execution time is less than the WCET estimate, the makespan is never worse.

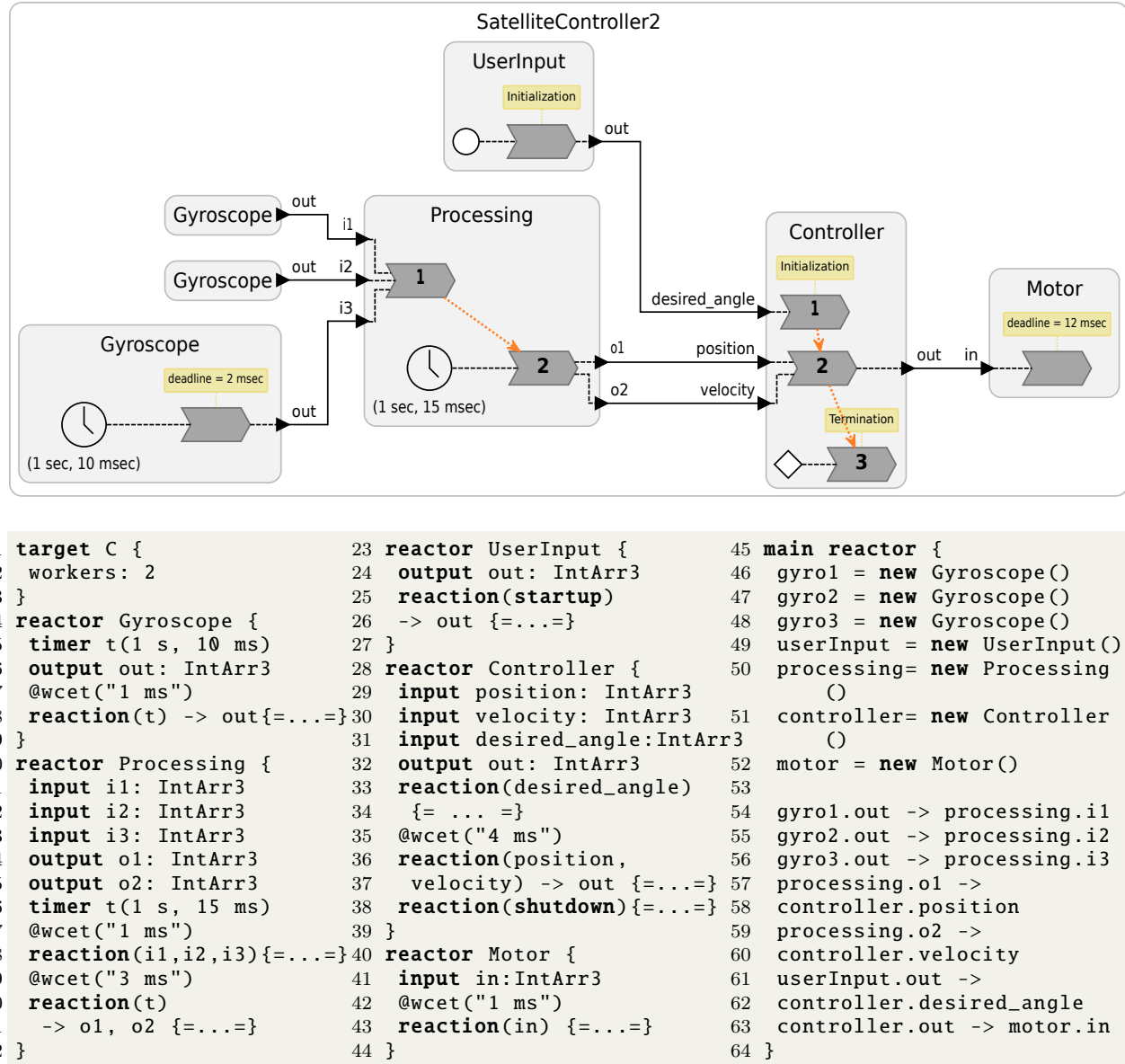


Figure 5.8: Satellite attitude controller with user input and termination, written in the synchronous subset of LF. The C code inside reactions is omitted for conciseness.

Generating SSFA from Reactors and Scheduling

To generate an SSFA from an LF program, the state space explorer identifies three execution phases of LF: *initialization*, *periodic*, and *shutdown*. The initialization phase contains a non-repeating sequence of state space nodes starting from the initial logical timestamp. The periodic phase contains a sequence of state space nodes that forms a hyperperiod. The

shutdown phase contains a sequence of nodes that follows after the periodic phase until the end of execution.

To help identify a hyperperiod for reactors, we first define the notion of a system *state*, as required by DTCM in Sec. 5.3. The definition is similar to the one given by Lin et al. [122].

Definition 6 (Reactor System State). *The system state s of an LF program is a tuple $s = (id^s, t^s, N, E)$, where $id^s \in ID$ is an identifier of the state, $t^s \in \mathcal{T}$ is a timestamp in a time domain, $N \subseteq \mathcal{N}$ is a set of reactions, and $E \subseteq \mathcal{E}$ is a set of pending events, where each event is $e = (id^e, t^e) \in E$ such that $id^e \in ID$ is an event identifier and $t^e \in \mathcal{T}$ is a timestamp.*

We now define the $\mathcal{H}$ function for reactors, as suggested in Sec. 5.3, to enable checking whether two states complete a hyperperiod.

Definition 7 (Hyperperiod Checking Function $\mathcal{H}$). *A sequence of states $(s_0, \dots, s_n)$ is said to form a hyperperiod, if the subsequent state s_{n+1} satisfies the following:*

1. $N_0 = N_{n+1}$. *This means that the same sets of reactions should be invoked;*
2. $\forall (id_0^e, t_0^e) = e_0 \in E_0. [\forall (id_{n+1}^e, t_{n+1}^e) = e_{n+1} \in E_{n+1}. [id_0^e = id_{n+1}^e \wedge t_0^e - t_0^s = t_{n+1}^e - t_{n+1}^s]]$. *This means that for all pending events in E_0 and E_{n+1} , not only must their event identifiers be the same, but their relative time offsets to the current timestamps must also be the same.*

If so, $\mathcal{H}(s_0, s_{n+1}) = \text{true}$, and $\mathcal{H}(s_0, s_{n+1}) = \text{false}$ otherwise.

The LF state space explorer implements Algorithm 1 from [122], which constructs the initialization and periodic phase by unrolling a simulated execution from the initial startup events until either:

1. there are no more pending events, *i.e.* $E = \emptyset$,
2. or a hyperperiod is found based on $\mathcal{H}$ defined in Def. 7.

When this procedure is completed, the sequence of states simulated typically forms a lasso shape. The explorer factors out the nodes leading up to the loop, *i.e.* the stem of the lasso, into the initialization phase, and the loop nodes into the periodic phase.

To show the above state exploration in action, in Fig. 5.8, we augment the running example with a **UserInput** reactor, which sends a desired satellite orientation to the controller when the LF program starts up. In addition, the **Controller** reactor has a third reaction triggered by a **SHUTDOWN** trigger, representing a termination procedure when the system shuts down. Fig. 5.9 now shows the three phases identified by the state space explorer with conditions for switching between phases. To construct a proper SSFA from this representation, a DAG must be constructed from each hyperperiod, a task performed by the DAG generator, which we discuss next.

Converting Reactor Phases to DAGs.

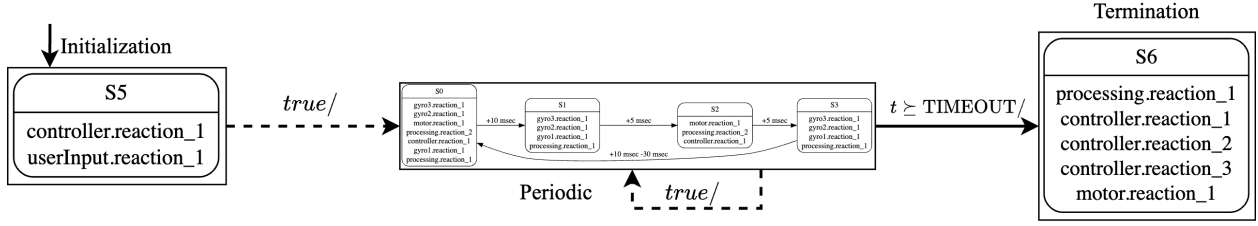


Figure 5.9: State space exploration identifies phases of the updated satellite controller (Fig. 5.8) with guarded transitions. The full-size periodic phase is shown in Fig. 5.2.

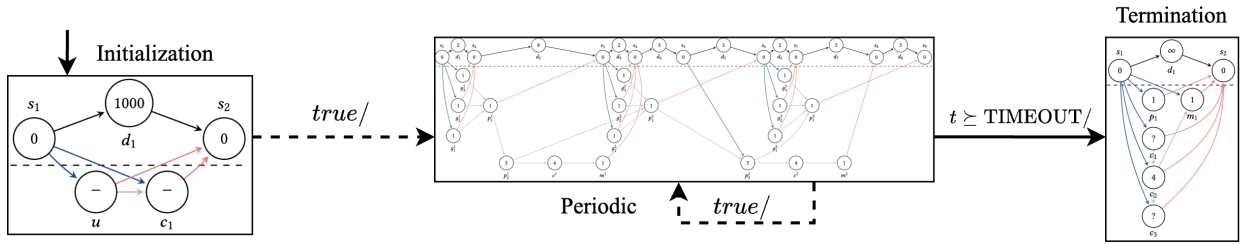


Figure 5.10: SSFA of the updated satellite controller generated from Fig. 5.9. In the Initialization mode, task nodes without WCETs are labeled “-”. In the Termination mode, dummy nodes of “ ∞ ” indicate that tasks do not have timing constraints on completion.

For each phase of the LF program, a DAG is constructed based on Algorithm 2. The DAG generator traverses each state in a phase, generates a sync node (introduced in Sec. 5.3) from its logical timestamp, and generates a task node for each reaction invocation in the state.

Edges are drawn between task nodes and sync nodes based on the reactor semantics, including:

1. Release edges from sync nodes marking release times to released task nodes;
2. Deadline edges that connect task nodes to sync nodes that mark their completion deadlines;
3. Data dependency edges between nodes having data dependency constraints.

The DAG generator produces additional sync nodes marking deadline on task completion. Lastly, it adds dummy nodes between sync nodes and uses additional edges to connect all sync nodes to form the virtual path. For example, the DAG in Fig. 5.4 is generated based on the rules above from Fig. 5.2. The resulting unscheduled SSFA after generating DAGs for all LF phases is shown in Fig. 5.10.

Algorithm 2 Generate a DAG from an LF Phase

```

1: procedure GENERATEDAG(phase)
2:    $dag = (V, E) \leftarrow \{\emptyset, \emptyset\}$ 
3:    $state \leftarrow phase.head$ 
4:   while True do
5:     BREAKIFALLVISITED(state, phase)
6:      $time \leftarrow GETLOGICALTIME(state)$ 
7:      $syncNode \leftarrow CREATESYNCNODE(time)$ 
8:      $V \leftarrow V \cup \{syncNode\}$ 
9:      $reactions \leftarrow GETREACTIONS(state)$ 
10:     $reactionNodes \leftarrow CREATETASKNODES(reactions)$ 
11:     $V \leftarrow V \cup reactionNodes$ 
12:     $E \leftarrow E \cup CREATESYNCTOTASKEDGES(syncNode, reactionNodes)$ 
13:     $E \leftarrow E \cup CREATEDATADEPENDENCYEDGES(reactions)$ 
14:     $state \leftarrow GETNEXTSTATEFROMPHASE(phase, state)$ 
15:  end while
16:   $V \leftarrow V \cup CREATEFINALSYNCNODE(phase, dag)$ 
17:   $E \leftarrow E \cup CREATEEDGESFORUNCONNECTEDTASKNODES(dag)$ 
18:  for all  $reaction \in GETREACTIONNODESWITHDEADLINES()$  do
19:     $deadlineNode = CREATEDEADLINESYNCNODE(reaction)$ 
20:     $V \leftarrow V \cup \{deadlineNode\}$ 
21:     $E \leftarrow E \cup CREATETASKTODEADLINEEDGE(reaction, deadlineNode)$ 
22:  end for
23:  ADDANDCONNECTDUMMYNODESBETWEENSYNCS(dag)
24:  return dag
25: end procedure

```

Scheduling each DAG by Partitioning. In our extended LF compiler, we support two different types of DAG schedulers: a load-balanced scheduler (LB) and an edge generation scheduler (EGS), by Sun et al. [179]. The load-balanced scheduler takes the generated DAG as its input and produces graph partitions that aim to assign each processor core the same amount of work measured in execution times. On the other hand, in case of the edge generation scheduler, instead of distributing workloads fairly across workers, the focus is on satisfying timing constraints. Given a DAG task set with a deadline, which can be the end of a hyperperiod, EGS checks whether the DAG task set is *trivially schedulable* on the given number of processors. If it is not trivially schedulable by construction, EGS adds edges to make it trivially schedulable. Once all DAGs in the SSFA are partitioned, the SSFA is considered scheduled and is ready to be compiled into an executable.

Table 5.1: Intermediate Representation for Composing Executable Schedules

Instruction	Description
<i>Basic instructions</i>	<i>Inspired by RISC-V [188]</i>
ADD op1, op2, op3	Add an integer variable (op2) and an integer variable (op3) and store the result in a destination variable (op1).
ADDI op1, op2, op3	Add an integer variable (op2) and an immediate (op3) and store the result in a destination variable (op1).
BEQ op1, op2, op3	Take the branch (op3) if the op1 variable value is equal to the op2 variable value.
BGE op1, op2, op3	Take the branch (op3) if the op1 variable value is greater than or equal to the op2 variable value.
BLT op1, op2, op3	Take the branch (op3) if the op1 variable value is less than the op2 variable value.
BNE op1, op2, op3	Take the branch (op3) if the op1 variable value is not equal to the op2 variable value.
JAL op1 op2	Store the return address to op1 and jump to a label (op2).
JALR op1, op2, op3	Store the return address to op1 and jump to a base address (op2) + an immediate offset (op3).
<i>Timing instructions</i>	<i>Inspired by PRET Machines [119, 197, 79]</i>
DU op1, op2	Delay until the physical clock reaches a base timepoint (op1) plus an offset (op2).
WLT op1, op2	Wait until a variable value (op1) is less than a desired value (op2).
WU op1, op2	Wait until a variable value (op1) is greater than or equal to a desired value (op2).
<i>Task instructions</i>	<i>Inspired by the reactor model [127]</i>
EXE op1, op2	Execute a function (op1) with argument (op2).
STP	Stop the execution.

Compiling Scheduled SSFA to Executable Schedules

In this section, we detail how a scheduled SSFA is further compiled into executable schedules. The executable schedules produced here are not directly emitted as native code; instead, they are expressed in an instruction-level intermediate representation (IR) and *interpreted at runtime* by PRETVM [121], a thin virtual machine layer that executes these schedules with analyzable worst-case execution time. The IR we develop in this subsection is precisely the PRETVM instruction set: each instruction listed in Table 5.1 is a PRETVM opcode that the virtual machine dispatches on its worker threads.

PRETVM’s Intermediate Representation (IR). From the partitioned DAGs, the schedule generator generates executable schedules, which are composed from an intermediate representation (IR) we design to direct how a CPU core should execute a DAG partition. The approach of using an IR for implementing real-time scheduling was pioneered by Henzinger et al. [69], from which we take inspiration. Table 5.1 shows the full IR, which is the PRETVM [121] instruction set for composing executable schedules. The IR combines elements from RISC-V [188], timing instructions from PRET Machines [119, 197, 79], and the reactor model’s task abstraction. At runtime, the PRETVM interpreter fetches and exe-

cutes these instructions on each worker thread, providing a uniform execution substrate for quasi-statically scheduled programs.

Synchronization. In our execution strategy, we support two types of synchronization: (i) *local synchronization* between a pair of workers, and (ii) *global synchronization* across all workers. Local synchronization is achieved by letting a **WU** block on another worker’s **counter** variable. On the other hand, global synchronization across all workers is performed at the completion of an SSFA state.

5.5 Evaluation

In this section, we evaluate the performance of executing quasi-statically scheduled LF programs and compare the execution overhead with that from using the dynamic scheduler of LF. Furthermore, we demonstrate the analyzable worst-case makespan property in Sec. 5.3 using a case study on a satellite attitude control system.

Performance

We evaluate the performance of the quasi-static scheduler using a set of LF benchmarks on a Raspberry Pi 4 Model B running the QNX (SDP 8.0) real-time operating system. By performance, we refer to the runtime overhead, quantified as *lags* in the context of LF. Specifically, lags are measured as the difference between a reaction’s actual physical firing time and its intended logical firing time. This metric applies to both the dynamic and the quasi-static scheduler. We compare the quasi-static schedulers’ performance against that of the default dynamic scheduler. The eight benchmarks are selected from [139, 122] with a focus on concurrent actor design patterns [76], which can be found in modern cyber-physical systems.

Table 5.2: Average, maximum, and standard deviation of the benchmark performance (lag) using the dynamic scheduler (DY), the static LOAD BALANCED scheduler (LB), and the static EDGE GENERATION scheduler (EGS).

Program	LoC (LF)	Average (μ s)			Maximum (μ s)			Standard Deviation (μ s)			PIR	
		DY	LB	EG	DY	LB	EG	DY	LB	EG	DY/LB	DY/EG
Philosophers [139]	314	86.1	2.99	2.53	1.37e+04	24.8	24	841	1.34	1.21	28.8	34.0
PingPong [139]	124	20.3	1.27	1.15	1.38e+04	6.43	5.8	377	0.758	0.677	16.0	17.7
Throughput [139]	166	5.21	2.43	2.53	10.1	6.33	7.43	1.38	1.19	1.25	2.1	2.1
ThreadRing [139]	217	6.95	2.29	2.02	13.1	5.89	5.15	2.39	1.32	1.17	3.0	3.4
LongShort	31	1.5e+06	77.2	78.7	2.92e+06	1.92e+03	1.95e+03	8.11e+05	379	386	19430.1	19059.7
CoopSchedule [80]	54	8.15	2.12	2.46	12.9	4.52	5.98	2.87	0.955	1.17	3.8	3.3
Counting [139]	179	5.81	1.41	1.27	98.5	7.76	9.2	3.3	0.592	0.703	4.1	4.6
ADASModel [122]	91	5.87	1.07	2.15e+03	22.2	3.35	1.2e+04	3.13	0.791	3.57e+03	5.5	0.0

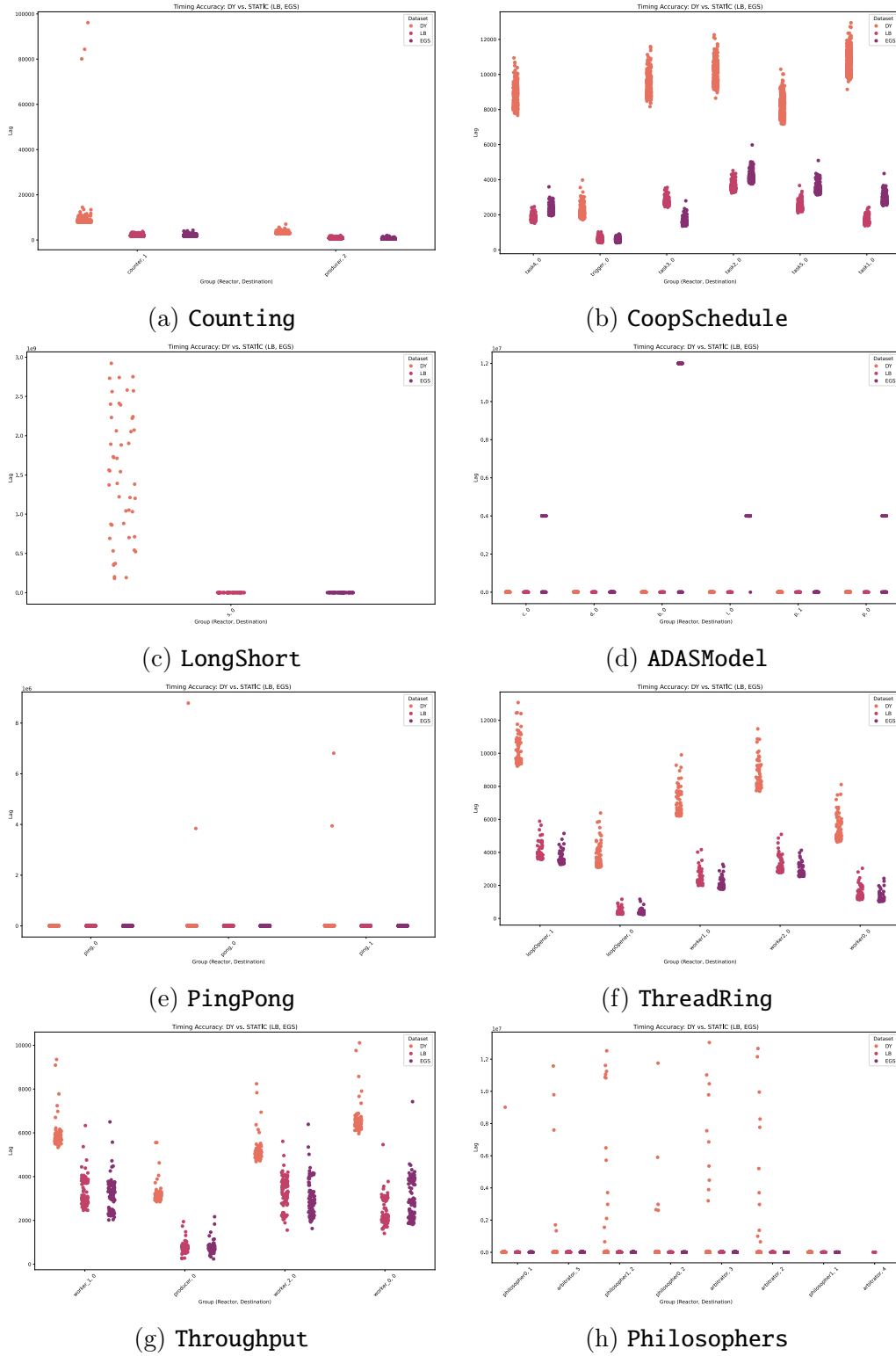


Figure 5.11: Timing accuracy results showing runtime overhead, measured on QNX (SDP 8.0) running on a Raspberry Pi 4. The X-axis represents reaction names, with each reaction displaying three distributions: DY, LB, and EGS. The Y-axis indicates lag in nanoseconds.

Table 5.2 shows an overview of the performance results. Compared to the dynamic scheduler, both static schedulers have smaller average lag, maximum lag, and standard deviation on all benchmarks, except for the **ADASModel** when scheduled with EGS. This outlier is due to edge generation’s scheduling strategy involving adding edges to the DAG, which could delay the releasing of tasks in exchange for schedulability guarantees. We note that the goal of using EGS is not to minimize lags, but to use its analysis capability to ensure that a DAG is schedulable.

Using the performance data reported in the “Average (μ s)” column of Table 5.2, we compute the Performance Improvement Ratio (PIR), shown in the “PIR” column of the same table. The PIR quantifies the relative performance of the baseline techniques LB and EG compared to DY. LB and EG consistently perform better than DY across various programs, with significant speedups. We compute the average PIR of LB and EG across all benchmarks while excluding **LongShort** (because it appears overly favorable to our approach). The resulting average PIR values indicate that LB and EG improve performance by factors of 9.0 and 9.3, respectively, compared to DY.

Fig. 5.11 shows strip plots of lags during the periodic phase of each program. For most reactions, LB and EGS have smaller lags than the dynamic scheduler.

Notably, for the **LongShort** benchmark, LB and EGS significantly outperform the dynamic scheduler. This performance gain can be explained by the fact that the quasi-static schedulers keep track of the current logical time for *each* reactor, meaning that every reactor can advance its logical time independently as long as the core LF semantics is respected, *i.e.* each reactor processes events in timestamp order. On the other hand, the dynamic scheduler uses a single variable to track logical time for *all* reactors, ensuring that all reactors advance time together. The **LongShort** benchmark has a design pattern that combines infrequent, long-running reactions with frequent, short-running reactions. The dynamic scheduler struggles because it performs a *global barrier synchronization* at the end of each timestamp. The barrier makes reactions that are to occur at the next timestamp wait on the long-running reactions, even if there are no more actual data dependencies to fulfill. In the quasi-static schedulers, the capability to advance time individually for each reactor effectively prevents waiting unnecessarily, resulting in significantly reduced lags.

Case Study: Satellite Attitude Control System

We return to the satellite attitude control system from Fig. 5.1 and demonstrate the use of our analyzable worst-case makespan property in Sec. 5.3 to perform compile-time validation of LF deadlines. The experiment is performed on FLEXPRET [197], a time-predictable processor, using the WCET analysis tool PLATIN [137] to obtain verifiable hard real-time guarantees.

Determining WCET Bounds. In LF, the business logic expressed in the reactions often consists of short, linear code sequences with bounded loops. These sequences are, therefore,

Table 5.3: WCET bounds for the reactions in the satellite controller computed by PLATIN.

Reaction	WCET (μs)
Gyro ₁	21
SensorFusion ₁	61
Controller ₂	172
Motor ₁	1

analyzable by existing WCET analysis tools including static tools such as PLATIN [163] and hybrid tools such as GAMETIME [170]. We compute WCET bounds for all reaction bodies.

Table 5.3 shows the WCET bounds computed by PLATIN for the reactions part of the periodic phase of the program, when targeting a FLEXPRET processor clocked at 100 MHz.

Scheduling. The reactions are annotated with their WCET bounds, allowing the EGS scheduler to produce a partitioned schedule. EGS rejects a program if the input DAGs are not schedulable. The LF compiler takes the partitioned DAG and produces a quasi-static schedule. A subtlety is that the EGS scheduler does not consider runtime overhead when scheduling the DAG; this means that for certain high-utilization programs with very tight WCET bounds, the resulting schedule might not be feasible if it results in too much coordination overhead. The LF compiler thus performs a final deadline satisfaction check, including the runtime overhead of the executable schedules as computed by PLATIN.

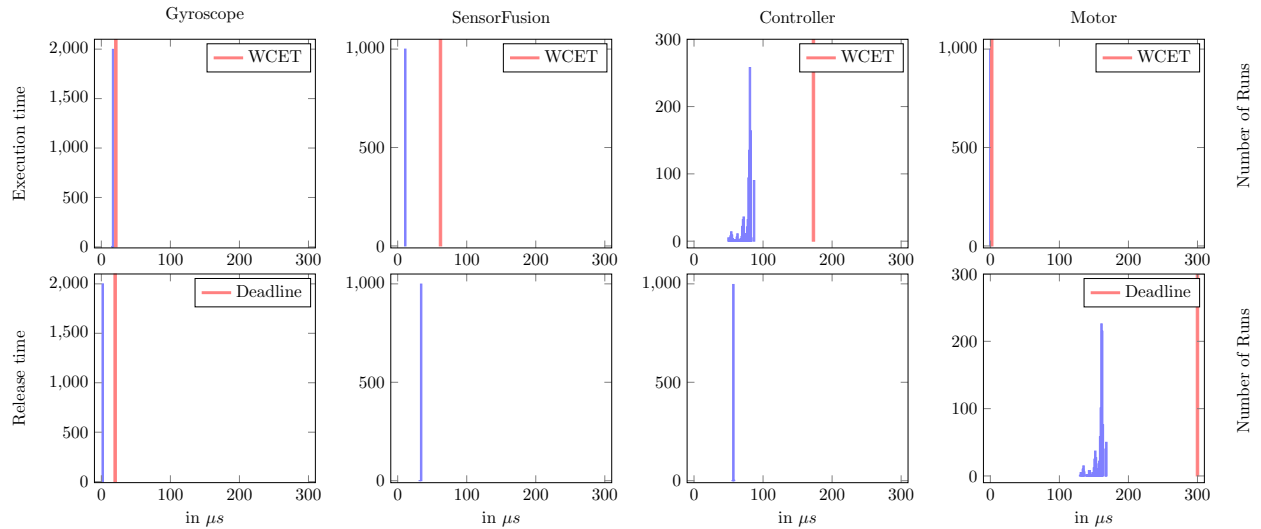


Figure 5.12: Measured release times and execution times of the reactions from Fig. 5.8 on FLEXPRET.

Evaluation. The resulting cross-compiled program targets a 100 MHz FLEXPRET processor supporting four fine-grain multithreading hard real-time threads, representing logical execution cores. A cycle-accurate emulation executes 1000 iterations of the program with simulated sensor inputs, and the execution times and completion times of each reaction are recorded. The results are shown in Fig. 5.12. The figure shows the measured execution time and the release time of each reaction in the periodic phase, represented as histograms. For the execution time, the WCET computed by PLATIN is shown by a red vertical line. For the release times, the LF release deadline, if applicable, is also shown by a red vertical line.

We observe that the measured execution times fall within the WCET bounds computed by PLATIN. As guaranteed by our methodology, all reactions meet their release deadlines. The release time of the Gyroscope reactions is at 17 μ s with zero variability, and the release time of the Motor reaction is met with a comfortable margin.

5.6 Design Space Exploration for Heterogeneous Platforms

Reactive cyber-physical systems (CPSs) have stringent real-time constraints and other practical considerations, such as energy, memory, heat, size, cost, etc., making them particularly challenging to design. Barring other considerations, it is hard to ensure that real-time constraints alone can be satisfied. Thus, a designer typically resorts to a *bottom-up* design flow, as noted by Henzinger and Kirsch [69], in which an implementation is first produced based on intuition, and then it undergoes testing. If the timing behavior violates constraints, the designer revises the implementation and tests it again. The process repeats until no violations are observed. This bottom-up process can be time-consuming and error-prone, since a code change fixing one constraint might violate others. In contrast, a *top-down* approach checks if such constraints can be satisfied at design time; only then is a correct-by-construction implementation generated. This reduces the time to market and ensures correctness of the design. Top-down approaches are widely adopted in the EDA domain, where a circuit's timing, area, and power can be predicted in the design tool, making it feasible to coordinate billions of transistors reliably. In the CPS domain, top-down approaches exist for models such as SDF [106]. Yet, for emerging programming models like LINGUA FRANCA (LF) [130], which facilitates the design and implementation of deterministic, real-time, and concurrent systems, providing top-down methods addressing timing and energy remains an open problem.

Energy consumption is an important dimension to consider, as many mission-critical CPSs are powered by batteries. For resource-constrained systems such as satellites, drones, and sensor networks, new techniques for energy management become ever more important due to recent trends toward adopting heterogeneous multicore platforms. Even for systems that are not considered resource-constrained in the traditional view, such as self-driving cars, energy management is increasingly a concern due to the deployment of energy-hungry

hardware, *e.g.*, GPUs.

In this section, we extend the quasi-static scheduling approach of the LINGUA FRANCA compiler [121] by exploring performance-energy trade-offs using MOCASIN [140], a design space exploration (DSE) framework. The extension takes an LF program with worst-case execution time (WCET) annotations, and outputs application logic and quasi-static schedules mapping LF reactions to a heterogeneous hardware platform.

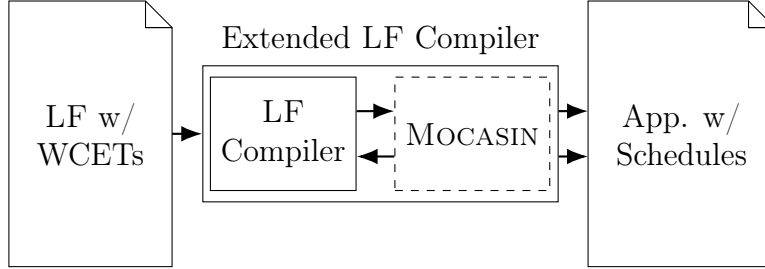


Figure 5.13: The proposed DSE tool flow. Dashed boxes denote external tools we build interfaces for.

MOCASIN is a DSE framework [140] that provides a comprehensive platform model capturing heterogeneous hardware on a relatively high abstraction level, supports applications in various input formats and using SDF or Kahn process network (KPN), implements various heuristics and meta-heuristics known from the literature, and provides a high-level simulator that allows estimating the makespan and the energy consumption of a given application on the target hardware when using a specific mapping. Fig. 5.13 shows the high-level tool flow, where the LF compiler interfaces bidirectionally with MOCASIN to explore design space alternatives.

DSE for LF based on PRETVM

Our approach relies on several key assumptions: (a) The chosen hardware is supported by MOCASIN. (b) The user has the ability to obtain *estimates* on the WCETs of C functions. Most of the time, the estimates could be obtained through direct measurements repeated a sufficient number of times. For certain platforms, it is possible to use commercial-off-the-shelf timing analysis tools, such as AbsInt [182]. We treat these input execution times as estimates, instead of the ground truth. This is sufficient for proving the property that “the program can provably meet the hard real-time constraints, *given that the estimates hold at runtime.*” (c) The LF program under analysis can be executed under LF’s fast mode, meaning that reactions do not wait for physical time points to be released. (d) The LF program only contains logically periodic tasks, *i.e.* driven by LF timers (same assumption as in [121]).

Tasks	Mapping 0	Mapping 1	Mapping 2
c1.reaction_1	Core 7 (A15)	Core 6 (A15)	Core 2 (A7)
c2.reaction_1	Core 6 (A15)	Core 6 (A15)	Core 2 (A7)
c3.reaction_1	Core 4 (A15)	Core 5 (A15)	Core 2 (A7)
d.reaction_2	Core 4 (A15)	Core 4 (A15)	Core 2 (A7)
d.reaction_3	Core 7 (A15)	Core 4 (A15)	Core 2 (A7)
d.reaction_4	Core 5 (A15)	Core 4 (A15)	Core 2 (A7)
Exec. time (s)	41.914	83.828	312.753
Total energy (J)	257.458	348.985	783.102

Table 5.4: A subset of mappings for the periodic phase generated by MOCASIN.

Preparing Inputs. The user first prepares an LF program with worst-case execution time information using the `@wcet` attribute. When targeting heterogeneous platforms such as the Odroid-XU4 (which uses Arm big.LITTLE architecture with two different types of CPU cores), the `@wcet` annotation contains multiple values—one per core type.

Generating and Registering Mappings. The compiler parses and transforms the input program into an abstract syntax tree (AST). A model of the LF program is created from the AST and fed into a state space explorer, which simulates and represents the program state space as a set of state space diagrams (SSDs) [122]. The SSDs are further transformed into a set of directed acyclic graphs (DAGs), each corresponding to a phase of LF execution (e.g., initialization, periodic, and shutdown phase). The role of the quasi-static schedulers is to assign tasks to workers by partitioning the DAGs [121].

In this extension, we implement an SDFG generator, which generates synchronous dataflow graphs (SDFGs) from input DAGs by drawing an additional edge from the tail node of the DAG to the head node. The generated SDFGs are then sent to a quasi-static scheduler based on MOCASIN, which invokes the `pareto_front` command on the SDFG for the periodic phase. MOCASIN runs DSE and outputs a mapping file enumerating a list of candidate mappings between tasks and CPU cores, which MOCASIN projects to yield the best timing and energy performance. The user can choose from the candidate mappings an ideal mapping, e.g., one projected to consume the least energy, and register the mapping into the original LF program.

Table 5.4 shows a subset of mappings that MOCASIN generated after design space exploration. In the generated file, each mapping maps a task to a particular core, with execution time and total energy estimated by MOCASIN.

Generating and Executing Schedules. Once the selected mapping is registered back into the LF program, the scheduler pools all tasks mapped to the same CPU core into one partition for a worker. Then for each worker, the LF compiler combines the worker’s

workload across different execution phases, forming a single quasi-static schedule for the worker. From each quasi-static schedule, a sequence of virtual instructions is generated for the worker [121].

As a result of compilation, LF generates programs and useful utilities that come from the original inputs. Then the user can pin different workers to the corresponding cores based on the user-selected mapping, which can be achieved on Linux by setting CPU Affinity and using the `cpuset` infrastructure.

Case Study: Energy-Aware Satellite Attitude Control

For this case study, we develop a realistic satellite attitude control application in LF, which has two parts: the first is a real-time reaction wheel controller based on [33]; the second is a sporadic file compression service, which compresses data files before they are sent back to earth. Each data file is further assumed to have a maximum size. In this case study, we focus on the file compression service, shown in Fig. 5.14.

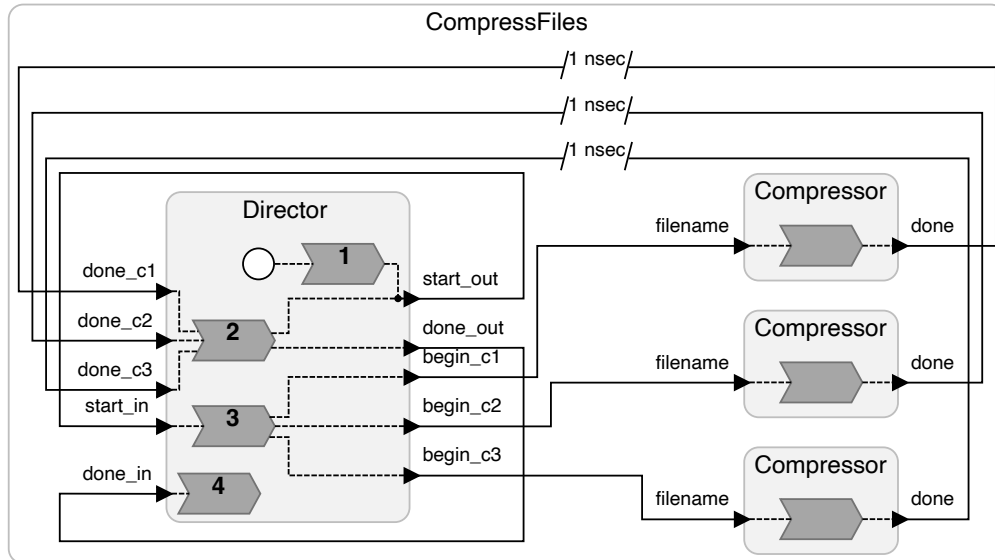


Figure 5.14: LF diagram of the file compression service.

The `CompressFiles` service consists of three **Compressor** reactors that implement the primary compression logic. In addition, there is a **Director** reactor that dispatches incoming jobs to one of the three **Compressor** reactors. Within the **Director** reactor, reactions 2 and 3 are of particular interest. Reaction 2 collects all **done** signals from the three **Compressor** reactors and determines whether processing can be stopped, in which case the **done_out** signal is set, otherwise the **start_out** signal is set. Reaction 3 reacts to the start signal described above and pushes a new set of files to be compressed to the **Compressor** reactors.

Experimental Setup. We run the experiments on the Hardkernel Odroid-XU4 platform featuring an Exynos 5422 big.LITTLE chip with four Cortex-A15 (“big”) and four Cortex-A7 (“LITTLE”) cores. The frequency ranges are set to 200 MHz to 2.0 GHz for big cores, and 200 MHz to 1.4 GHz for LITTLE cores. We measured the energy consumption of the Odroid-XU4 board using a ZES Zimmer LMG450 Power Analyzer connected to DC input with an external readout rate of 20 Sa/s.

The main method of controlling the power consumption of a CPU is frequency scaling, apart from shutting down individual cores. Linux implements a concept called *governors*, which implement different strategies for scaling the CPU frequency. The relevant governors we evaluate are:

- **Performance** Sets the frequency to the highest possible.
- **Powersave** Sets the frequency to the lowest possible.
- **Ondemand** Scales the frequency based on the current load.
- **Schedutil** Scales the frequency based on the CPU utilization info from the scheduler, and uses Energy Aware Scheduling (EAS) to map tasks to cores to minimize energy consumption on heterogeneous platforms.

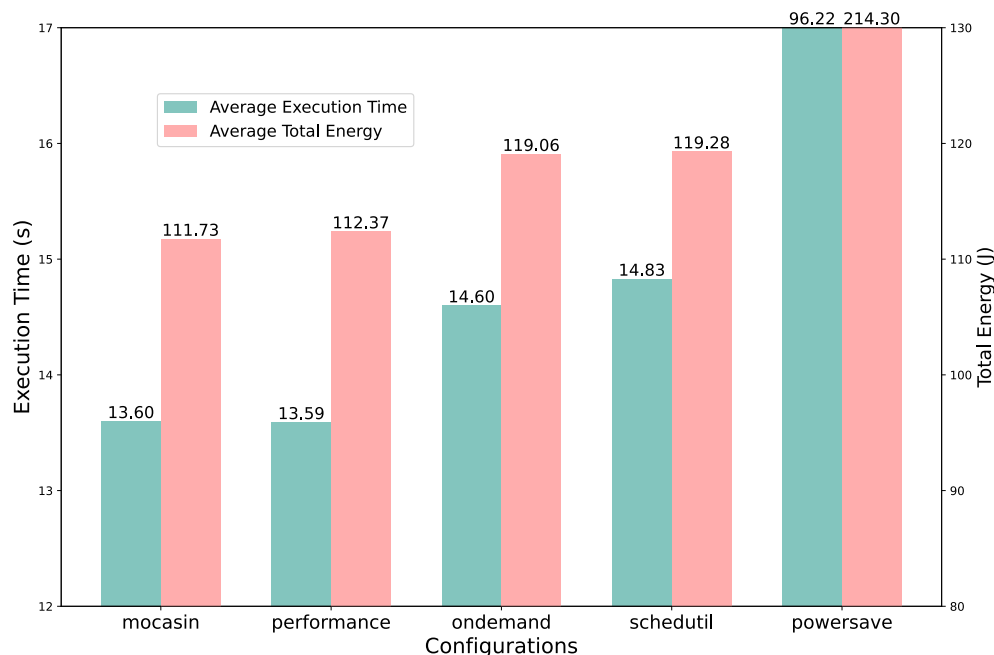


Figure 5.15: Execution time and total energy measured.

Measurement Results. We perform five sets of experiments using the LF program **CompressFiles**. The first experiment uses the quasi-static scheduler with a task-to-core mapping computed by MOCASIN, and the rest of the experiments use LF with the default dynamic scheduler coupled with the Linux governors, representing the common strategies a regular Linux user would use to minimize energy. For each configuration, the service compresses 15 CSV files, and measurements were taken based on the average of one hundred repeated program executions. In the first experiment, MOCASIN explores the design space assuming constant CPU frequencies; to satisfy this assumption, the **performance** governor is used. In addition, we pin reactions to each type of CPU core under a fixed frequency and use LF’s tracing utility to collect the reactions’ core-specific WCET estimates.

Table 5.4 shows a subset of mappings generated by MOCASIN in the first experiment. We choose to use mapping 0 as it is projected to consume a relatively small amount of energy among all available mappings without compromising execution time.

Fig. 5.15 shows the measured execution time and total energy from all five experiments. While the **powersave** governor does exhibit the least power (in terms of wattage), since it sets each CPU core to the minimum frequency (200 MHz), it significantly prolongs the execution time (96.22 s) and, in fact, yields the *largest* total energy consumption (214.3 J) among the experiments. This result highlights the fact that saving power is not the same as saving total energy.

The **ondemand** and **schedutil** governors (the latter of which uses EAS) have similar execution times and total energy. They perform better than **powersave** since they dynamically scale the CPU frequencies based on the workload. The shorter execution times result in less total energy consumed. MOCASIN and **performance** turn out to deliver the best performance in the end, since big cores are prioritized in both strategies, resulting in two of the smallest execution times and lowest total energy consumption values. This could seem counter-intuitive to regular users who choose **powersave** or **schedutil** to conserve energy.

Discussion. While using big cores seems to minimize execution time and total energy, the choice of cores depends on additional factors. In the space domain, these factors include mission specifications, solar power charging rate, and battery size. Here, we focus on the *total* (static + dynamic) energy consumption, instead of the *dynamic* energy alone, on the assumption that the hardware on the satellite enters a low-power or non-consuming state after completing a computational task, rendering static energy consumption negligible in this context. If the mission restricts low-power mode, the **performance** governor (using big cores) would not be ideal due to high static energy consumption. A low solar power charging rate and small battery may also make big cores impractical. In such cases, trading execution time for power efficiency with LITTLE cores might be better. A top-down design technique, such as our proposed LF extension, could better handle such design complexities as design space exploration could factor in these constraints upfront. On the other hand, bottom-up approaches could be error-prone, as we have seen with the **powersave** case, and inefficient, as they often require trial and error.

5.7 Conclusion and Future Work

In this chapter, we have presented the first unified quasi-static scheduling approach based on an intermediate formalism called state space finite automata (SSFA), bridging the gap between high-level MoCs and low-level executable schedules. We show that our unified scheduling approach applies to existing models, including SDF, BDF, SADF, and LET, and we identify a class of deterministic timed concurrent models (DTCMs) to which the SSFA-based methodology is applicable. We demonstrate the application of the proposed approach to an emerging MoC—the reactor model. Our evaluation confirms that the SSFA-based quasi-static scheduling approach applies successfully to reactors, reducing average runtime overhead by $9\times$ compared to the default dynamic scheduler. It also adds compile-time deadline validation to the LF compiler, benefiting from the properties of quasi-static scheduling. We further extend the quasi-static scheduling technique with support for interfacing with MOCASIN, which enables DSE for a subset of LF programs on heterogeneous platforms and generates schedules that encode user-specified trade-offs between execution time and total energy consumption. Moving forward, we plan to evaluate our approach quantitatively using larger-scale benchmarks and to apply our unified SSFA-based scheduling framework to more emerging models of computation.

Chapter 6

Timing Analysis of Sequential Programs using GAMETIME

“When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind.”

—Lord Kelvin, *“Electrical Units of Measurement,”* 1883.

6.1 Background

The system-level timing analysis in Chapter 2 treats each reaction’s execution time as a given quantity, either measured from a trace or bounded by a worst-case estimate. In practice, obtaining that quantity is itself a difficult problem: the execution time of a sequential program depends on the input data, the control-flow path taken through the code, and microarchitectural effects such as caching and pipelining. This chapter describes GAMETIME [168, 169, 170], a toolkit for predicting the execution time of such a program along any path, and in particular for estimating its worst-case execution time (WCET). The WCET estimates produced by GAMETIME can serve directly as the execution-time parameters consumed by Proposition 2.1 of Chapter 2.

The WCET problem. A terminating software task may take exponentially many control-flow paths through its code. Measuring the execution time of every path is infeasible, and purely static approaches that model the micro-architecture must be ported anew whenever the target platform changes [168]. GAMETIME sidesteps both problems by using SMT-generated tests to measure a *small, carefully chosen* subset of basis paths on the actual target platform, and by applying game-theoretic online learning to infer a timing model that predicts the timing of every other path.

Game-theoretic online learning. GAMETIME leverages a measurement-driven, game-theoretic learning algorithm to efficiently analyze the timing of sequential programs. After selecting basis paths, GAMETIME uses SMT-based test generation to produce inputs that force the program to execute along those paths. These tests are then run on the target platform, repeatedly and potentially from different start states, to obtain timing measurements. The game-theoretic online learning component uses these measurements to learn a timing model of the platform, represented as a weighted graph over the program’s control-flow structure. Once learned, this model predicts the timing of paths that were not directly measured and supports the search for worst-case execution paths.

Control-flow graph and DAG representation. GAMETIME operates on programs whose loops have statically known bounds and whose functions can be inlined. It first extracts the control-flow graph (CFG) of the sequential program under analysis, unrolls every loop to its maximum bound, and inlines all function calls, producing a *directed acyclic graph* (DAG) $G = (V, E)$ with a single source s (entry point) and a single sink t (exit point) [168, 30]. Each source-to-sink path in this DAG corresponds to a unique execution path of the original program. A path can be represented as a 0-1 vector $\mathbf{p} \in \{0, 1\}^{|E|}$, where the i -th component is 1 if and only if edge e_i lies on the path.

Basis paths and the basis matrix. The set of all source-to-sink path vectors spans a subspace of $\mathbb{R}^{|E|}$, and a basis for this subspace can be found with at most $|E|$ paths [169, 170]. These *basis paths* have two key properties: (i) every source-to-sink path $\mathbf{p}$ can be written as a linear combination $\mathbf{p} = \sum_b c_b \mathbf{p}_b$ of the basis paths, and (ii) the coefficients satisfy $|c_b| \leq 2$ for all b , ensuring that the timing of every path is of the same order of magnitude as the timings of the basis paths [30]. The number of feasible basis paths is bounded by $|E| - |V| + 2$, which is far smaller than the (potentially exponential) total number of paths.

Stacking the b basis-path vectors as rows yields the *basis matrix* $B = [\mathbf{p}_1, \dots, \mathbf{p}_b]^\top$, a $b \times m$ matrix where $m = |E|$. Each column of B corresponds to an edge of the DAG, and each row corresponds to a basis path. To find a good basis, GAMETIME iteratively replaces rows of B so as to *maximize* $|\det(B)|$, the absolute value of the determinant of the basis matrix [90, 169]. Geometrically, maximizing the determinant is equivalent to maximizing the volume of the parallelepiped spanned by the basis-path vectors, which spreads the vertices of the spanned polytope as far apart as possible [170]. A basis with this property is called a *2-barycentric spanner*: it guarantees bounded coefficients $|c_b| \leq 2$ and hence bounded prediction error. In practice, each candidate replacement row is found by solving an integer linear program that maximizes the subdeterminant, and its feasibility is then checked by an SMT solver (Section 6.3 describes the KLEE-based implementation of this check).

Example: modular exponentiation. The basis path concept is illustrated by Kotker et al. [90] using a modular exponentiation routine that computes $\text{base}^{\text{exponent}} \bmod p$ via repeated squaring. When the loop is unrolled for a 2-bit exponent, the resulting DAG

has $m = 9$ edges and four feasible source-to-sink paths, one for each value of `exponent` $\in \{0, 1, 2, 3\}$. Each path is encoded as a 0-1 vector of length 9; for instance, edges 2 and 3 correspond to the else-branch and then-branch of the first conditional (`exponent & 1`)`== 1`, respectively. Three of the four paths, say $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ (corresponding to `exponent` values 1, 2, and 0), form a basis. The fourth path $\mathbf{p}_4$ (corresponding to `exponent` = 3) is then expressible as $\mathbf{p}_4 = \mathbf{p}_1 + \mathbf{p}_2 - \mathbf{p}_3$, as shown in Figure 6.1. More generally, the unrolled DAG for an N -bit exponent has an “ N -diamond” structure with 2^N feasible paths but at most $N + 1$ basis paths, so the number of measurements grows only linearly with N .

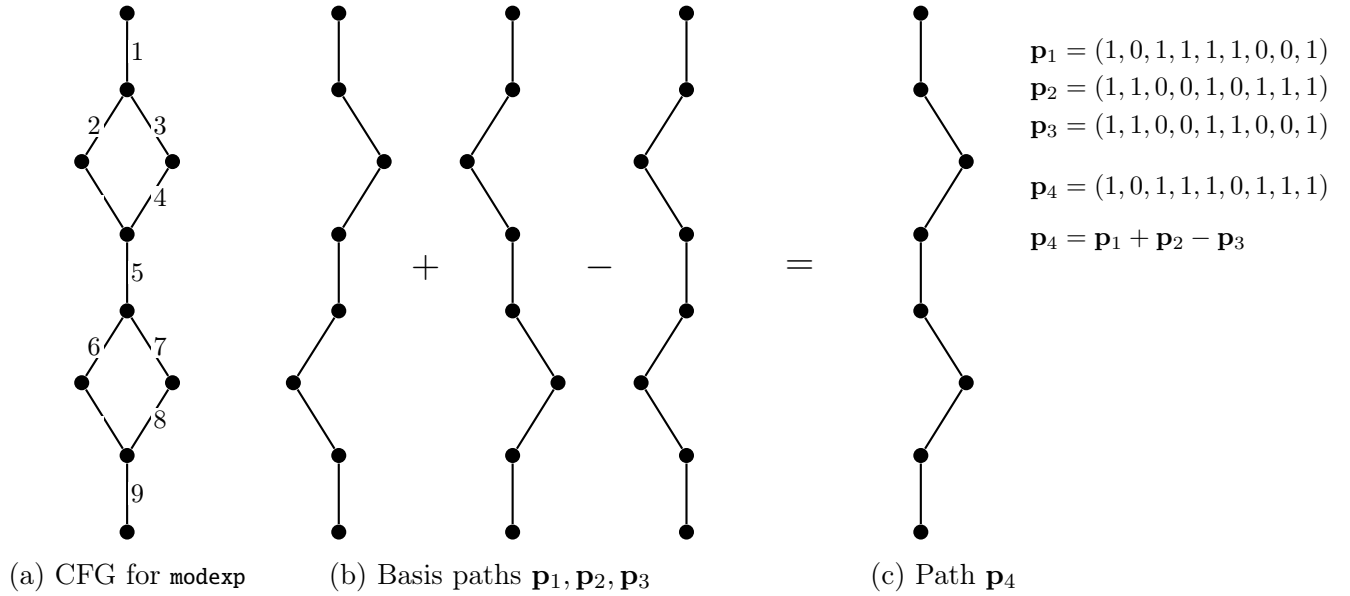


Figure 6.1: CFG and basis paths for the unrolled modular exponentiation code with a 2-bit exponent (from Kotker et al. [90]). (a) The full DAG with 9 edges. (b) Three basis paths $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$. (c) The fourth path $\mathbf{p}_4$ expressed as $\mathbf{p}_1 + \mathbf{p}_2 - \mathbf{p}_3$.

Feasibility checking and test generation. Not every source-to-sink path in the DAG is *feasible*: some paths are prevented by correlations among branch conditions (e.g., two branches that test the same variable). As GAMETIME constructs each candidate basis path, it queries a satisfiability modulo theories (SMT) solver to determine whether there exists a concrete input that drives execution along that path [168, 90]. If the solver returns SAT, the satisfying assignment is used as the test input; if UNSAT, the path is *infeasible* and is discarded, and a new candidate is selected.

Measurement and prediction. Once a feasible basis has been computed, the program is compiled for the target platform and executed on each basis-path test case. The end-to-end execution time of each basis path is recorded in a vector $v = (v_1, v_2, \dots, v_b)$, where v_i is the

measured execution time of the i -th basis path. GAMETIME then estimates a weight vector $\tilde{w} \in \mathbb{R}^m$, with one weight per edge, by solving the linear system

$$B \tilde{w} = v, \quad (6.1)$$

where B is the $b \times m$ basis matrix defined above [90]. Since Algorithm 1 of Kotker et al. [90] ensures that the determinant of B is never zero, B is invertible (when $b = m$) or admits a Moore–Penrose pseudoinverse B^+ , yielding $\tilde{w} = B^+ v$. Intuitively, $\tilde{w}_e$ estimates the execution time contribution of edge e , so the predicted execution time of any path $\mathbf{p}$ is simply $\hat{\ell}_{\mathbf{p}} = \tilde{w}^\top \mathbf{p}$, the dot product of the weight vector with the path vector. Under a bounded-repeatability assumption, namely that the actual execution time of any path deviates from its expected value by at most $\mu_{\max}$ cycles, Bundala and Seshia [30] show that the estimation error is bounded by $2k \mu_{\max}$, where k is a quantity that depends only on the graph structure and the choice of basis, not on the measurements themselves.

ILP-based extremal path extraction. To find the longest (worst-case) or shortest (best-case) path, GAMETIME solves an integer linear program (ILP) [30]. The ILP assigns a non-negative weight w_e to each edge, constrained so that the total weight of each measured basis path matches its observed execution time (up to tolerance D). Binary flow variables encode the selection of a source-to-sink path, and the objective maximizes (or minimizes) the total weight along the selected path. If the path returned by the ILP is infeasible, a constraint blocking that path ($\sum_{e \in \pi} b_e < |\pi|$) is added and the ILP is re-solved. When the unsatisfiable core of the infeasibility proof is available, a tighter constraint is added using only the edges in the core [30], pruning multiple infeasible paths at once.

Overall flow. In summary, the GAMETIME analysis proceeds as follows:

1. Extract the CFG from the sequential program under analysis and convert it to a DAG by unrolling loops and inlining functions.
2. Compute a set of feasible basis paths using linear algebra on the DAG, with an SMT solver to check and enforce feasibility.
3. Compile the program for the target platform and measure the execution time of each basis-path test case.
4. Formulate and solve an ILP to extract the extremal (e.g., longest) path. Check the candidate path for feasibility using an SMT solver; if infeasible, add a blocking constraint to the ILP and re-solve. Repeat until a feasible extremal path is found.
5. Return the extremal path, its predicted execution time, and the concrete test input that exercises it.

This measurement-based approach makes GAMETIME portable across platforms: only step 3 interacts with the target hardware, so switching platforms requires only re-measuring the basis paths. Extensions to handle data-dependent instruction timing and to formalize the notion of timing repeatability are described by Wasson [187].

6.2 A New GAMETIME with LLVM Support

The original toolchain. The GAMETIME 1.5 implementation described by Kotker et al. [90] relied on two compiler front-ends that are no longer maintained. First, the CIL (C Intermediate Language) framework [145] preprocessed the C source by inlining functions, unrolling loops, converting `switch` statements to nested `if` statements, and normalizing pointer expressions. Second, the Microsoft Phoenix compiler front-end extracted the control-flow graph from the CIL-processed code and provided a C# API for traversing the resulting intermediate representation in static single assignment (SSA) form. GAMETIME then performed its own lightweight symbolic execution over the Phoenix IR to collect path conditions, translated them into SMT queries in the `QF_AUFBV` logic (quantifier-free arrays, uninterpreted functions, and bit-vectors), and dispatched those queries to Z3 or Boolector. Integer linear programs were solved through the PuLP Python package, interfacing with solvers such as GLPK, Gurobi, or IBM CPLEX.

Motivation for a new implementation. Both CIL and Microsoft Phoenix have been discontinued, making GAMETIME 1.5 increasingly difficult to build and maintain on modern systems. More fundamentally, both front-ends were specific to C, whereas many safety-critical embedded applications are now written in C++ or are generated from higher-level languages. The LLVM compiler infrastructure [99] provides a natural replacement: Clang compiles C and C++ to LLVM intermediate representation (IR), and front-ends for Rust, Swift, and other languages target the same IR. By operating on LLVM IR rather than a language-specific representation, a new GAMETIME implementation can, in principle, analyze any program that compiles to LLVM IR.

Architecture of the new GAMETIME. The new implementation, publicly available at <https://github.com/icyphy/gametime>, replaces both CIL and Phoenix with LLVM and Clang (version 16) while preserving the core GAMETIME algorithm. The main changes are as follows:

- *Preprocessing.* Function inlining and loop unrolling are performed using LLVM’s built-in optimization passes and custom LLVM passes, rather than CIL’s OCaml-based source-to-source transformations. The result is an LLVM IR bitcode file in which all loops have been fully unrolled and all callees have been inlined.

- *CFG extraction.* The control-flow graph is extracted directly from the LLVM IR bitcode, producing a DAG in the same format used by the original GAMETIME algorithm. Nodes correspond to basic blocks and edges to branches.
- *Path feasibility checking.* Instead of GAMETIME’s custom symbolic execution over the Phoenix IR, the new implementation delegates feasibility checking to KLEE [32], a mature symbolic execution engine that operates natively on LLVM IR. This change eliminates the need for GAMETIME to maintain its own expression language and SMT encoding; the details are described in Section 6.3.
- *ILP solving.* The PuLP-based formulation is retained, using the GNU Linear Programming Kit (GLPK) as the default solver.
- *Measurement backends.* The new GAMETIME supports multiple measurement backends, including cycle-accurate emulation on FLEXPRET [197] (a RISC-V processor designed for mixed-criticality systems), as well as direct measurement on x86 and ARM hosts.

6.3 Checking Path Infeasibility with KLEE

As described in Section 6.1, GAMETIME must determine whether a candidate path through the DAG is *feasible*, i.e., whether there exists a concrete input that drives execution along exactly that path. In GAMETIME 1.5, this check was performed by a custom symbolic execution engine that translated path conditions from the Microsoft Phoenix IR into SMT queries and dispatched them to Z3 or Boolector. The new LLVM-based GAMETIME replaces this entire subsystem with KLEE [32], a symbolic execution engine that operates directly on LLVM IR bitcode and uses STP or Z3 as its constraint solver.

The challenge is that KLEE, by default, explores *all* reachable paths through a program. GAMETIME, however, needs to answer a more targeted question: is there an input that follows *one specific* path? To bridge this gap, GAMETIME instruments the LLVM IR with *condition variables* that constrain KLEE to the target path.

Condition variable instrumentation. Let $\pi = \langle B_0, B_1, \dots, B_{k-1} \rangle$ be the sequence of basic blocks along the candidate path, and let $\bar{\pi} = \langle B_k, B_{k+1}, \dots, B_{N-1} \rangle$ be the remaining basic blocks that lie in the function but *not* on the path. GAMETIME inserts a global Boolean variable `conditional_var_i` for each basic block and instruments the LLVM IR bitcode as follows:

- For every on-path block B_i ($0 \leq i < k$), the variable `conditional_var_i` is initialized to `false`, and a store instruction `conditional_var_i = true` is inserted at the beginning of B_i .

- For every off-path block B_j ($k \leq j < N$), the variable `conditional_var_j` is initialized to `true`, and a store instruction `conditional_var_j = false` is inserted at the beginning of B_j .

After the instrumentation, a condition variable holds the value `true` if and only if the corresponding block was *visited when it should be* (on-path) or *not visited when it should not be* (off-path). Therefore, the conjunction $\bigwedge_{i=0}^{N-1} cv_i$, where cv_i denotes `conditional_var_i`, is `true` exactly when execution follows path π and no other.

Example: checking feasibility of path $\mathbf{p}_1$. We illustrate the full procedure using the modular exponentiation example from Section 6.1. Recall that the unrolled DAG has two branch points, corresponding to the two evaluations of the conditional `(exponent & 1) == 1`, and four basic blocks at these branches (two then-blocks and two else-blocks). Path $\mathbf{p}_1$ (corresponding to `exponent = 1`, binary `01`) takes the then-branch at the first conditional and the else-branch at the second.

Figure 6.2 shows the unrolled `modexp` function annotated with the condition variable assignments that GAMETIME inserts for path $\mathbf{p}_1$. The function has $N = 8$ basic blocks in total, and path $\mathbf{p}_1$ visits $k = 6$ of them. Each on-path block receives a variable initialized to `false` that is set to `true` upon entry; each off-path block receives a variable initialized to `true` that is set to `false` upon entry. The reader can trace path $\mathbf{p}_1$ through the listing by following the blocks whose annotations read `cv_i = true`: execution enters the function (`cv_0`), takes the first then-branch because `exponent & 1` is 1 (`cv_1`), performs the multiplication (`cv_2`), shifts the exponent and squares the base (`cv_3`), skips the second then-branch because the shifted exponent is even (`cv_4`), and returns (`cv_5`). The two off-path blocks (the first else-branch and the second then-branch) are never entered, so their variables `cv_6` and `cv_7` remain at their initial value of `true`. All eight variables end up `true`, confirming feasibility.

KLEE harness generation. In addition to the instrumented function, GAMETIME generates a wrapper `main()` that declares the function parameters as local variables, marks them symbolic with `klee_make_symbolic()`, calls the function under analysis, and asserts that every condition variable is `true`. Figure 6.3 shows the generated harness for the modular exponentiation example. The compiled LLVM bytecode of this program (with the `store` instructions inserted into each basic block of `modexp` by the `modify_bitcode.cpp` LLVM pass) is then handed to KLEE for symbolic execution.

KLEE execution and output. KLEE explores the symbolic execution tree of the instrumented program. On each explored path, it checks whether all eight `klee_assert` calls succeed. If an assertion fails, KLEE records an `.assert.err` file alongside the corresponding `.ktest` file and continues exploring other paths. A `.ktest` file *without* a matching `.assert.err` file represents an input for which all assertions passed, meaning the target path is feasible.

Figure 6.4 shows the actual output produced by the `ktest-tool` utility when decoding a `.ktest` file from a feasibility check on the `modexp` benchmark. In this run, KLEE found that

```

1 int modexp(int base, int exponent) {
2   int result = 1;
3   // cv_0 = true (on-path: entry)
4   if ((exponent & 1) == 1) {
5     // cv_1 = true (on-path: 1st then)
6     result = (result * base) % p;
7   } else {
8     // cv_6 = false (off-path: 1st else)
9   }
10  // cv_2 = true (on-path: after 1st branch)
11  exponent >>= 1;
12  base = (base * base) % p;
13  // cv_3 = true (on-path: before 2nd branch)
14  if ((exponent & 1) == 1) {
15    // cv_7 = false (off-path: 2nd then)
16    result = (result * base) % p;
17  } else {
18    // cv_4 = true (on-path: 2nd else)
19  }
20  // cv_5 = true (on-path: return)
21  exponent >>= 1;
22  base = (base * base) % p;
23  return result;
24 }

```

Figure 6.2: Unrolled `modexp` annotated with condition variable assignments for path p_1 (`exponent = 1`). Variables `cv_0` through `cv_5` (on-path) are initialized to `false` and set to `true`; `cv_6` and `cv_7` (off-path) are initialized to `true` and set to `false`. When `exponent = 1`, all variables end up `true`.

`base = 0` and `exponent = 124 (0x7c)` satisfy all assertions, confirming that the candidate path is feasible. GAMETIME extracts these values and uses them as the concrete test input to measure the execution time of the path on the target platform.

If, instead, the candidate path were infeasible (e.g., a path that takes the then-branch at both conditionals but with an `exponent` that is even), every test case generated by KLEE would have at least one assertion failure, and no `.ktest` file without a corresponding `.assert.err` would exist. GAMETIME would then add a blocking constraint to the ILP (Section 6.1) and extract the next candidate path.

Implementation. The instrumentation is carried out in two stages. First, a Python script (`to_klee_format.py`) rewrites the C source to include the KLEE header, declare the condition variables, and generate the wrapper `main()`. Second, an LLVM pass written in C++

```
1 #include <klee/klee.h>
2 #include <stdbool.h>
3
4 // on-path (6 blocks)
5 bool cv_0 = false;
6 bool cv_1 = false;
7 bool cv_2 = false;
8 bool cv_3 = false;
9 bool cv_4 = false;
10 bool cv_5 = false;
11 // off-path (2 blocks)
12 bool cv_6 = true;
13 bool cv_7 = true;
14
15 int main() {
16     int base, exponent;
17     klee_make_symbolic(&base, sizeof(base), "base");
18     klee_make_symbolic(&exponent, sizeof(exponent), "exponent");
19     modexp(base, exponent);
20     klee_assert(cv_0); klee_assert(cv_1);
21     klee_assert(cv_2); klee_assert(cv_3);
22     klee_assert(cv_4); klee_assert(cv_5);
23     klee_assert(cv_6); klee_assert(cv_7);
24     return 0;
25 }
```

Figure 6.3: Generated KLEE harness for checking feasibility of path p_1 . The six on-path condition variables are initialized to `false`; the two off-path variables are initialized to `true`. The wrapper makes `base` and `exponent` symbolic, calls `modexp`, and asserts all eight variables are `true`.

```

ktest file : '.../klee-last/test000006.ktest'
args : ['.../modexp_..._gvMod.bc']
num objects: 2
object 0: name: 'base'
object 0: size: 1
object 0: data: b'\x00'
object 0: hex : 0x00
object 0: int : 0
object 0: uint: 0
object 0: text: .
object 1: name: 'exponent'
object 1: size: 1
object 1: data: b'|'
object 1: hex : 0x7c
object 1: int : 124
object 1: uint: 124
object 1: text: |

```

Figure 6.4: Actual `ktest-tool` output from a feasibility check on the `modexp` benchmark. KLEE found concrete inputs `base = 0` and `exponent = 124` that drive execution along the candidate path. Paths are abbreviated with `...` for readability.

(`modify_bitcode.cpp`) operates on the compiled bytecode to insert the `store` instructions into the correct basic blocks, using the LLVM `IRBuilder` API. The labels of the basic blocks along the target path are extracted from the LLVM IR by matching the `%n:` block-label syntax, providing the mapping between DAG edges and LLVM basic blocks.

6.4 Applications of GAMETIME in this Thesis

The work presented in this thesis aims to verify timing properties of real-time concurrent systems, primarily in the context of LINGUA FRANCA. Since GAMETIME targets sequential programs, it is infeasible to directly use GAMETIME to obtain a system-level WCET bound on the LF-generated `main()` function in C, since the generated code uses multithreading (not supported by GAMETIME) and, due to the nature of embedded systems, may not terminate at all. However, the timing analysis of sequential programs enabled by GAMETIME nicely complements two system-level analyses presented earlier in this dissertation, so that system-level timing guarantees can be obtained by using GAMETIME to analyze sequential reactions.

System-level timing analysis (Chapter 2). Proposition 2.1 developed in Chapter 2 expresses the physical firing time of each reaction as a max-plus linear function of input tags, timestamps, execution times, and network latencies. A central parameter in this formulation

is $e_n(k)$, the worst-case execution time of reaction n at tag k . When the goal is to predict timing behavior of a future run, for instance to check whether deadlines can be violated under worst-case conditions, these execution times must be estimated *before* the system runs. GAMETIME can supply these estimates by analyzing each reaction body as sequential C code, producing a WCET bound together with the worst-case input that achieves it. Plugging these bounds into Proposition 2.1’s recurrence yields a conservative prediction of physical firing times and, in turn, of the consistency, availability, and latency trade-offs for the distributed system.

Quasi-static scheduling (Chapter 5). The quasi-static scheduling framework in Chapter 5 compiles a LINGUA FRANCA program into a DAG of tasks, where each task v_i is a non-preemptive reaction with WCET C_i . The scheduler uses these WCETs to compute earliest and latest start times, to verify deadline satisfaction, and to determine whether the DAG is trivially schedulable on a given number of processors. Obtaining tight WCET bounds is therefore essential: overly pessimistic bounds waste processor time and may cause the scheduler to reject programs that are in fact schedulable, while optimistic bounds risk deadline violations at run time. GAMETIME can analyze each reaction to produce tight, measurement-based WCET estimates, closing the loop between the scheduling framework and the actual timing behavior of the generated code.

6.5 Future Work

Several directions remain open for improving and extending the GAMETIME approach.

Exploiting UNSAT cores to prune infeasible paths. When KLEE determines that a candidate path is infeasible, the current implementation adds a single blocking constraint that excludes exactly that path from future ILP solutions (Section 6.1). In many cases, however, only a small subset of the edges along the path are mutually incompatible. If the underlying SMT solver can produce an *unsatisfiable core*, i.e., a minimal set of constraints whose conjunction is already unsatisfiable, then a tighter blocking constraint can be added that rules out all paths containing that subset of edges [30]. This would prune multiple infeasible paths at once and significantly reduce the number of ILP-KLEE iterations required to find a feasible extremal path. Integrating UNSAT core extraction into the KLEE-based workflow is a key next step for improving the scalability of the new GAMETIME.

Measurement-based network latency analysis. The core idea behind GAMETIME, namely that a small number of carefully chosen measurements can predict the behavior of exponentially many scenarios, is not limited to execution times of individual sequential programs. In a distributed cyber-physical system, network latencies exhibit analogous variability due to routing, congestion, and protocol-level effects that are difficult to capture in a

fine-grained analytical model. A natural generalization is to apply the basis-path methodology to network paths: select a basis set of end-to-end routes, measure their latencies on the actual network, and use the measurements to predict the latency of every other route. Preliminary work in this direction [8] combines GAMETIME with the LINGUA FRANCA coordination language to derive end-to-end timing guarantees for federated (distributed) LF programs, including network delays. Extending this approach to a full network-level timing analysis framework is a promising avenue for future research.

Deep-learning-based timing models. GAMETIME already constructs a learning-based timing model: it repeatedly measures basis paths, potentially from different start states, and uses those measurements to infer a weighted graph model for predicting the timing of other paths. However, there is a limitation in the expressiveness of the learned model. In the standard GAMETIME formulation, the prediction is linear in a path vector, with timing represented by scalar edge weights. This additive model may be too coarse to capture data-dependent microarchitectural effects such as cache behavior and branch prediction, where the cost of a block can depend on the execution context. Deep learning offers a natural way to learn a richer, nonlinear model from repeated path measurements and associated path or state features. Investigating whether deep learning models can improve prediction accuracy while retaining the measurement efficiency of the basis-path approach is an open question.

Chapter 7

Conclusion

“The future cannot be predicted, but futures can be invented.”

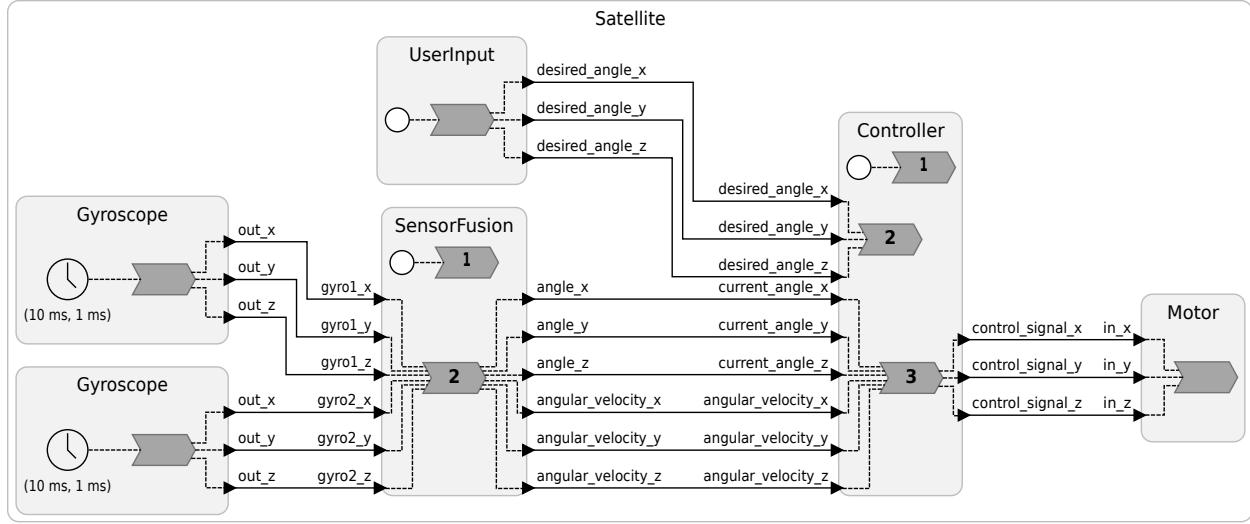
—*Dennis Gabor, Inventing the Future, 1963.*

7.1 Future Work

My future agenda aims to automate the proposed framework in Figure 1.1, scale up its capabilities across design and verification, and generalize to intriguing new domains. Agenda item 1 details the steps for full automation, with an illustration shown in Figure 7.1. Agenda item 2 enhances the framework’s analytical capabilities. Agenda item 3 aims to automate unbounded verification using invariant learning and assume-guarantee reasoning. Agenda items 4–6 extend the framework to the Robotic Operating System (ROS), time-predictable networks, and deterministic machine learning.

1. Automated Neurosymbolic Framework

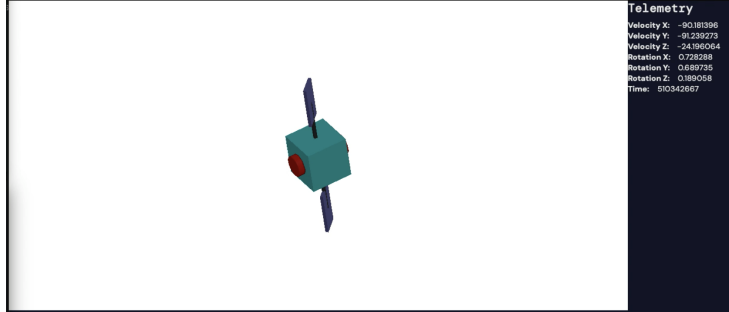
AI models like Claude and GPT-5-Codex have significantly increased programmers’ productivity. *Can CPS engineers also benefit from generative AI when tools are available to guarantee the reliability and performance of AI-generated designs?* Currently, the tools introduced thus far in Figure 1.1 are partially automated by the LINGUA FRANCA toolchain (particularly verification and refinement), or individually invoked by the user. The ultimate goal is to streamline the neurosymbolic workflow coordinating generative AI and symbolic tools. Using Model Context Protocol (MCP), tools such as Proposition 2.1 (Chapter 2), POLYVER [38], and the LF compiler [130] can be exposed as callable services with typed inputs and outputs. An AI model can therefore orchestrate the symbolic tools via MCP, iteratively improving the design based on tool feedback until it reaches a deployable and correct-by-construction implementation.



(a) AI generates a design, invokes analyzers, and iterates on analyzer feedback.

```
@property(
name="skip_invalid_input",
tactic="bmc", spec="G[0,11ms]
(!controller.input_valid
==>(motor.x == 0 && motor.y
== 0 && motor.z == 0)))")
```

(b) AI calls POLYVER to verify formal properties and iterates on failure.



(c) AI calls the LF compiler, iterates on failure, and deploys code in simulation.

Figure 7.1: Building a satellite reaction wheel controller [180, 38, 181] using the neurosymbolic framework in Figure 1.1

2. Learning-based and Distributed Timing Analyses

Computing the worst-case execution time (WCET) for a sequential code block is a classically hard problem. Traditional WCET analysis involves constructing a detailed hardware model, yet this approach is becoming intractable as modern hardware architectures grow in complexity. To remove the need to model hardware, measurement-based approaches emerged, such as GAMETIME [168], an approach based on the notion of “basis paths.” While GAMETIME is the first approach to timing analysis that uses machine learning, it uses a relatively simple learning framework, which may be limited in accuracy, as it learns a weighted model at the CFG level that can be too coarse to make accurate predictions. *Can we leverage deep learning to train more accurate models for predicting WCETs based on timing measurement*

data? In addition, as CPS tend to be distributed, *can we generalize this technique to a distributed, system-level design with many concurrently interacting code blocks?*

3. Automated Unbounded Verification

While the LF VERIFIER [122] and POLYVER [38] could reason about unbounded properties using the classic k-induction, this technique requires the invariant under verification to be *inductive*, which often involves manually strengthening the invariant, making the process unscalable and time-consuming. Property directed reachability (PDR) [24, 48] has received much attention in the formal methods community since it extends k-induction by iteratively strengthening an invariant until an inductive invariant is found. *Can we leverage algorithms like PDR to obtain unbounded guarantees for verifying CPS engineering models? Can assume-guarantee reasoning further accelerate this process?*

4. Robot Operating System

The Robot Operating System (ROS) has become the de facto middleware used in robotics research and has garnered a large community developing reusable packages for visualization, simulation, and control. However, ROS has severe limitations in predictability, as its runtime implementation permits nondeterministic event orderings and undersampled timer firings. On the other hand, LINGUA FRANCA (LF) is known for its determinism and predictability due to its logical-time semantics, but it lacks a large package ecosystem. *Can we combine LF and ROS to obtain predictable real-time behavior while leveraging ROS's extensive packages?*

5. Time-predictable Networks

Hard real-time guarantees have become all the more important for distributed CPS, where meeting one machine's deadlines depends critically on the timeliness of data sent from upstream machines. To ensure predictable timing in networks, an Ethernet standard called Time-Sensitive Networking (TSN) has been developed. However, it remains nontrivial to configure a TSN network to obtain verified timing guarantees. *Can we synthesize verifiable TSN schedules from engineering models?*

6. Deterministic Machine Learning

Machine learning has a well-known reproducibility problem, affecting popular architectures behind Large Language Models (LLMs) like Transformers. A study from Thinking Machine Lab shows that even when temperature is set to zero, LLM output can still be nondeterministic given the same input, because the training framework PyTorch by default does not process batched data in the same order, causing non-associative floating point operations to produce different outputs on each run [66]. This nondeterministic ordering problem shares

many similarities with the ordering problem solved by LINGUA FRANCA, and there is potential for using a deterministic abstraction to systematically address this problem, rather than hunting down nondeterministic loopholes on a case-by-case basis. *Can we define a concurrent model of computation for specifying deterministic ordering and data flow for machine learning pipelines? Can we code-generate high-performance CUDA implementations from the model?*

7.2 Final Remarks

This thesis identifies three toolchain gaps in CPS engineering: analysis for *logical time*, verification on CPS engineering models, and hard-real-time scheduling without manual tuning. *Can we design a toolchain that closes all three gaps in a unified workflow?* The answer developed here is not a single algorithm or tool, but an integrated toolchain: extend timing analysis to logical-time models, verify properties where requirements are stated, and compile verified models into implementations scheduled with analyzable, predictable timing behavior.

The common thread across the thesis is the separation between logical behavior and physical realization. Logical time, reactors, and LINGUA FRANCA give engineers a deterministic model of what a system should do. The CAL-based system-level timing analysis in Chapter 2 and the timing analysis of sequential programs in Chapter 6 expose how physical execution affects when that behavior occurs. The distinction between engineering and scientific models in Chapter 3, together with the LF VERIFIER in Chapter 4, shows how formal reasoning can be applied directly to the design artifact rather than to a separate, manually maintained verification model. Finally, the SSFA-based quasi-static scheduling framework in Chapter 5 demonstrates how analyzable models can be refined into deterministic, high-performing implementations.

A second lesson is that predictability need not require abandoning modern software practice. The techniques developed in this thesis keep reaction bodies in mainstream programming languages while keeping distributed coordination, multicore systems, heterogeneous platforms, and learning-driven, measurement-based timing analysis in view. The goal is not to replace existing engineering intuitions, but to augment those intuitions with crisp formal tools: critical paths derived from eigenvalues instead of intuition alone about bottlenecks, counterexamples from exhaustive formal analysis instead of errors that may slip through incomplete test coverage, schedulability checks instead of manual priority tuning, and tight WCET estimates learned from measurements instead of hand-written timing budgets.

Much remains to automate and scale, as the future-work agenda makes clear. If CPS models can be made executable, analyzable, verifiable, and compilable within the same framework, then reliability no longer has to arrive only after long cycles of implementation and testing. It can become part of the design process itself. That is the broader promise of verifiable cyber-physical systems: not merely to find errors earlier, but to make correct, deterministic, and high-performing systems an expected outcome of principled engineering practice.

Bibliography

- [1] Daniel Abadi. “Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story”. In: *Computer* 45.2 (2012), pp. 37–42. DOI: [10.1109/MC.2012.33](https://doi.org/10.1109/MC.2012.33).
- [2] Boussad Addad, Said Amari, and Jean-Jacques Lesage. “Networked Conflicting Timed Event Graphs Representation in (Max, +) Algebra”. In: *Discrete Event Dynamic Systems* 22.4 (2012), pp. 429–449. DOI: [10.1007/s10626-012-0136-0](https://doi.org/10.1007/s10626-012-0136-0). URL: <https://doi.org/10.1007/s10626-012-0136-0>.
- [3] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. Cambridge, MA: MIT Press, 1986.
- [4] Gul Agha. “Concurrent object-oriented programming”. In: *Communications of the ACM* 33.9 (1990), pp. 125–140.
- [5] Gul A. Agha. “Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems”. In: *Formal Methods for Open Object-based Distributed Systems, IFIP Transactions*. Ed. by E. Najm Stefani and J.-B. Chapman and Hall, 1997.
- [6] Jade Alglave, Luc Maranget, and Michael Tautschnig. “Herding cats: Modelling, simulation, testing, and data mining for weak memory”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.2 (2014), pp. 1–74.
- [7] Rajeev Alur. *Principles of cyber-physical systems*. MIT press, 2015.
- [8] Anagha Ananth et al. *Bridging Logical and Physical Time: Automated WCET Analysis for Lingua Franca Programs*. Final report for CS262A: Advanced Topics in Computer Systems, UC Berkeley. 2025. URL: https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F25/projects/reports/project1012_paper_ver2_54013958461773778657.pdf.
- [9] Charles André. *Syntax and Semantics of the Clock Constraint Specification Language (CCSL)*. Research Report RR-6925. INRIA, 2009, p. 37. URL: <https://hal.inria.fr/inria-00384077>.
- [10] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. “Bounded model checking of software using SMT solvers instead of SAT solvers”. In: *International Journal on Software Tools for Technology Transfer* 11.1 (2009), pp. 69–83.

- [11] Roy Armoni et al. “SAT-based Induction for Temporal Safety Properties”. In: *Electronic Notes in Theoretical Computer Science* 119.2 (2005), pp. 3–16. ISSN: 1571-0661. DOI: <https://doi.org/10.1016/j.entcs.2004.12.021>. URL: <https://www.sciencedirect.com/science/article/pii/S1571066105000873>.
- [12] F. Baccelli et al. *Synchronization and Linearity, An Algebra for Discrete Event Systems*. New York: Wiley, 1992.
- [13] Kyungmin Bae et al. “Verifying Ptolemy II Discrete-Event Models Using Real-Time Maude”. In: *Formal Methods and Software Engineering*. Ed. by Karin Breitman and Ana Cavalcanti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 717–736. ISBN: 978-3-642-10373-5.
- [14] Clark Barrett et al. “Satisfiability Modulo Theories”. In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Second. IOS Press, 2021. Chap. 33, pp. 1267–1329.
- [15] Soroush Bateni et al. “Risk and Mitigation of Nondeterminism in Distributed Cyber-Physical Systems”. 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, 2023, 1–11. ISBN: 9798400703188. DOI: [10.1145/3610579.3613219](https://doi.org/10.1145/3610579.3613219). URL: <https://doi.org/10.1145/3610579.3613219>.
- [16] Soroush Bateni et al. “Risk and Mitigation of Nondeterminism in Distributed Cyber-Physical Systems”. In: *ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2023. DOI: [10.1145/3610579.3613219](https://doi.org/10.1145/3610579.3613219).
- [17] Albert Benveniste and Gérard Berry. “The Synchronous Approach to Reactive and Real-Time Systems”. In: *Proceedings of the IEEE* 79.9 (1991), pp. 1270–1282.
- [18] Albert Benveniste et al. “The synchronous languages 12 years later”. In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83.
- [19] Béatrice Bérard et al. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.
- [20] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.
- [21] Armin Biere et al. “Bounded model checking.” In: *Handbook of satisfiability* 185.99 (2009), pp. 457–481.
- [22] Armin Biere et al. “Symbolic model checking without BDDs”. In: *International conference on tools and algorithms for the construction and analysis of systems*. Springer. 1999, pp. 193–207.
- [23] George E. P. Box and Norman R. Draper. *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Statistics. Wiley, 1987. ISBN: 0471-81033-9.

- [24] Aaron R. Bradley. “SAT-Based Model Checking without Unrolling”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87. ISBN: 978-3-642-18275-4.
- [25] Eric Brewer. “CAP twelve years later: How the "rules" have changed”. In: *Computer* 45.2 (2012), pp. 23–29. DOI: **10.1109/MC.2012.37**.
- [26] Eric Brewer. “Towards Robust Distributed System”. In: *Symposium on Principles of Distributed Computing (PODC)*. Keynote talk. 2000.
- [27] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. “Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions”. In: ed. by E. Brinksma and K. G. Larsen. LNCS 2404. 2002, pp. 78–92.
- [28] Joseph T. Buck. “Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model”. PhD thesis. EECS Department, University of California, Berkeley, 1993. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/1993/2429.html>.
- [29] J.T. Buck and E.A. Lee. “Scheduling dynamic dataflow graphs with bounded memory using the token flow model”. In: *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 1. 1993, 429–432 vol.1. DOI: **10.1109/ICASSP.1993.319147**.
- [30] Daniel Bundala and Sanjit A. Seshia. “On Systematic Testing for Execution-Time Analysis”. In: *CoRR* abs/1506.05893 (2015). arXiv: **1506.05893**. URL: <http://arxiv.org/abs/1506.05893>.
- [31] Alan Burns and Sanjoy Baruah. “Sustainability in real-time scheduling”. In: *Journal of Computing Science and Engineering* 2.1 (2008), pp. 74–97.
- [32] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*. USENIX Association, 2008, pp. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.
- [33] Janette Cardoso et al. “Lab on the Real-Time Control of Reaction Wheel”. Unpublished lab exercise for the course 1MAE803: Real-time Control of Aerospace Systems, M.Sc in Aerospace Engineering, ISAE-SUPAERO. 2022.
- [34] Christos G. Cassandras and Stéphane Lafortune. *Introduction to discrete event systems*. Springer, 2008.
- [35] Adam Cataldo et al. “A Constructive Fixed-Point Theorem and the Feedback Semantics of Timed Systems”. In: *Workshop on Discrete Event Systems (WODES)*. 2006.

- [36] Adrien Champion et al. “The Kind 2 model checker”. In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 510–517.
- [37] Ernest Chang and Rosemary Roberts. “An improved algorithm for decentralized extrema-finding in circular configurations of processes”. In: *Communications of the ACM* 22.5 (1979), pp. 281–283.
- [38] Pei-Wei Chen et al. *PolyVer: A Compositional Approach for Polyglot System Modeling and Verification*. 2025. arXiv: 2503.03207 [cs.PL]. URL: <https://arxiv.org/abs/2503.03207>.
- [39] Guy Cohen et al. “A Linear-System-Theoretic View of Discrete-Event Processes and Its Use for Performance Evaluation in Manufacturing”. In: *IEEE Transactions on Automatic Control* 30.3 (1985), pp. 210–220.
- [40] James C. Corbett et al. “Spanner: Google’s Globally-Distributed Database”. In: *ACM Tr. on Computer Systems (TOCS)* 31.8 (2013).
- [41] R.L. Cruz. “A calculus for network delay. I. Network elements in isolation”. In: *IEEE Transactions on Information Theory* 37.1 (1991), pp. 114–131. DOI: 10.1109/18.61109.
- [42] Raymond A. Cuninghame-Green. *Minimax Algebra*. Vol. 166. Lecture Notes in Economics and Mathematical Systems. Berlin, Heidelberg: Springer, 1979. ISBN: 978-3-540-09113-4. DOI: 10.1007/978-3-642-48708-8. URL: <https://doi.org/10.1007/978-3-642-48708-8>.
- [43] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.
- [44] Julien Deantoni et al. “Debugging and Verification Tools for Lingua Franca in GEMOC Studio”. In: *2021 Forum on specification & Design Languages (FDL)*. IEEE. 2021, pp. 01–08.
- [45] Julien Deantoni et al. *Operational Semantics of the Model of Concurrency and Communication Language*. Research Report RR-8584. INRIA, Sept. 2014, p. 23. URL: <https://hal.inria.fr/hal-01060601>.
- [46] Jack B. Dennis. *First Version Data Flow Procedure Language*. Report MAC TM61. MIT Laboratory for Computer Science, 1974.
- [47] Emanuele D’Osualdo, Jonathan Kochems, and C-H Luke Ong. “Automatic verification of Erlang-style concurrency”. In: *International Static Analysis Symposium*. Springer. 2013, pp. 454–476.
- [48] Niklas Een, Alan Mishchenko, and Robert Brayton. “Efficient implementation of property directed reachability”. In: *2011 Formal Methods in Computer-Aided Design (FMCAD)*. 2011, pp. 125–134.

- [49] Javier Esparza et al. “An SMT-based approach to coverability analysis”. In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 603–619.
- [50] Inc. Express Logic. “Measuring Real-Time Performance of an RTOS”. In.
- [51] Moritz Eysholdt and Heiko Behrens. “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*. ACM. 2010, pp. 307–309.
- [52] Emilia Farcas and Wolfgang Pree. “Hyperperiod bus scheduling and optimizations for TDL components”. In: *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)*. IEEE. 2007, pp. 1262–1269.
- [53] Miguel Felder, Dino Mandrioli, and Angelo Morzenti. “Proving properties of real-time systems through logical specifications and Petri net models”. In: *IEEE Transactions on Software engineering* 20.2 (1994), pp. 127–141.
- [54] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: *Journal of the ACM* 32.2 (1985), pp. 374–382. DOI: [10.1145/3149.214121](https://doi.org/10.1145/3149.214121). URL: <https://doi.org/10.1145/3149.214121>.
- [55] Anders Franzén. “Using satisfiability modulo theories for inductive verification of Lustre programs”. In: *Electronic Notes in Theoretical Computer Science* 144.1 (2006), pp. 19–33.
- [56] Brian Gallagher, Hidlehiko Akatsuka, and Hideaki Suzuki. “Wireless communications for vehicle safety: Radio link performance and wireless connectivity methods”. In: *IEEE Vehicular Technology Magazine* 1.4 (2006), pp. 4–24.
- [57] Hubert Garavel et al. “CADP 2011: a toolbox for the construction and analysis of distributed processes”. In: *International Journal on Software Tools for Technology Transfer* 15.2 (2013), pp. 89–107.
- [58] Marc C. W. Geilen et al. “Scenarios in Dataflow Modeling and Analysis”. In: *System-Scenario-based Design Principles and Applications*. Cham: Springer International Publishing, 2020, pp. 145–180. ISBN: 978-3-030-20343-6. DOI: [10.1007/978-3-030-20343-6_8](https://doi.org/10.1007/978-3-030-20343-6_8). URL: https://doi.org/10.1007/978-3-030-20343-6_8.
- [59] Kai-Björn Gemlau et al. “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 (2021), pp. 1–27. DOI: [10.1145/3381847](https://doi.org/10.1145/3381847).
- [60] A. H. Ghamarian et al. “Throughput analysis of synchronous data flow graphs”. In: *Proceedings - Design, Automation and Test in Europe, DATE* (2006), pp. 116–121. ISSN: 15301591. DOI: [10.1109/DATE.2008.4484672](https://doi.org/10.1109/DATE.2008.4484672).

- [61] A.H. Ghamarian et al. “Throughput Analysis of Synchronous Data Flow Graphs”. In: *Sixth International Conference on Application of Concurrency to System Design (ACSD’06)*. 2006, pp. 25–36. DOI: [10.1109/ACSD.2006.33](https://doi.org/10.1109/ACSD.2006.33).
- [62] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *ACM SIGACT News* (2002), 33(2). DOI: [10.1145/564585.564601](https://doi.org/10.1145/564585.564601).
- [63] Adwait Godbole, Yatin A Manerkar, and Sanjit A Seshia. “Automated conversion of axiomatic to operational models: Theory and practice”. In: *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2022*. 2022, p. 331.
- [64] George Hagen and Cesare Tinelli. “Scaling up the formal verification of Lustre programs with SMT-based techniques”. In: *2008 Formal Methods in Computer-Aided Design*. IEEE. 2008, pp. 1–9.
- [65] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. “Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE”. In: *IEEE transactions on software engineering* 18.9 (1992), pp. 785–793.
- [66] Horace He and Thinking Machines Lab. “Defeating Nondeterminism in LLM Inference”. In: *Thinking Machines Lab: Connectionism* (2025). DOI: [10.64434/tml.20250910](https://doi.org/10.64434/tml.20250910). URL: <https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/>.
- [67] Bernd Heidergott, Geert Jan Olsder, and Jacob van der Woude. *Max Plus at Work: Modeling and Analysis of Synchronized Systems: A Course on Max-Plus Algebra and Its Applications*. Princeton Series in Applied Mathematics. Princeton, NJ: Princeton University Press, 2006. ISBN: 978-0-691-11763-8.
- [68] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. “Giotto: A Time-Triggered Language for Embedded Programming”. In: *EMSOFT 2001*. Vol. LNCS 2211. Springer-Verlag, 2001, pp. 166–184.
- [69] Thomas A. Henzinger and Christoph M. Kirsch. “The embedded machine: Predictable, portable realtime code”. In: *International Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2002, pp. 315–326.
- [70] Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic. “Schedule-Carrying Code”. In: *Embedded Software*. Ed. by Rajeev Alur and Insup Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 241–256. ISBN: 978-3-540-45212-6.
- [71] Carl Hewitt. “Viewing control structures as patterns of passing messages”. In: *Journal of Artificial Intelligence* 8.3 (1977), pp. 323–363.
- [72] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Stanford, CA, USA, August 20-23, 1973*. 1973, pp. 235–245.

- [73] C. A. R. Hoare. “Towards a Theory of Parallel Programming”. In: *Operating Systems Techniques. Proceedings of a Seminar*. Ed. by C. A. R. Hoare and R. H. Perrott. Vol. 9. A.P.I.C. Studies in Data Processing. London: Academic Press, 1972, pp. 61–71.
- [74] Benjamin Horowitz. *Single-mode, single-processor Giotto scheduling*. Computer Science Division, University of California, 2003.
- [75] Tsung-Wei Huang et al. “Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.6 (2022), pp. 1303–1320. DOI: **10.1109/TPDS.2021.3104255**.
- [76] Shams M. Imam and Vivek Sarkar. “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, 2014, 67–80. ISBN: 9781450321891. DOI: **10.1145/2687357.2687368**. URL: **<https://doi.org/10.1145/2687357.2687368>**.
- [77] Mohammad Mahdi Jaghoori, Ali Movaghar, and Marjan Sirjani. “Modere: The model-checking engine of Rebeca”. In: vol. 2. Jan. 2006, pp. 1810–1815. DOI: **10.1145/1141277.1141704**.
- [78] Axel Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann, 2003.
- [79] Erling R. Jellum et al. “InterPRET: A Time-Predictable Multicore Processor”. In: *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. CPS-IoT Week ’23. San Antonio, TX, USA: Association for Computing Machinery, 2023, 331–336. ISBN: 9798400700491. DOI: **10.1145/3576914.3587497**. URL: **<https://doi.org/10.1145/3576914.3587497>**.
- [80] Erling Rennemo Jellum et al. “Beyond the Threaded Programming Model on Real-Time Operating Systems”. In: *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023)*. Ed. by Federico Terraneo and Daniele Cattaneo. Vol. 108. Open Access Series in Informatics (OASICS). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 3:1–3:13. ISBN: 978-3-95977-268-6. DOI: **10.4230/OASICS.NG-RES.2023.3**. URL: **<https://drops.dagstuhl.de/opus/volltexte/2023/17734>**.
- [81] Jeff C Jensen, Edward A Lee, and Sanjit A Seshia. “An introductory lab in embedded and cyber-physical systems”. In: *LeeSeshia. org, Berkeley, CA* (2012).
- [82] Gilles Kahn. “The Semantics of a Simple Language for Parallel Programming”. In: *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974, pp. 471–475.

- [83] Reinier van Kampenhout, Sander Stuijk, and Kees Goossens. “Programming and analysing scenario-aware dataflow on a multi-processor platform”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 876–881.
- [84] Ramtin Khosravi et al. “Actors Upgraded for Variability, Adaptability, and Determinism”. In: *Active Object Languages: Current Research Trends*. Ed. by Frank S. de Boer et al. Lecture Notes in Computer Science. Springer, 2024, pp. 226–260. DOI: **10.1007/978-3-031-51060-1_9**.
- [85] Christoph M Kirsch and Ana Sokolova. “The logical execution time paradigm”. In: *Advances in Real-Time Systems* (2012), pp. 103–120.
- [86] Christoph M Kirsch and Ana Sokolova. “The logical execution time paradigm”. In: *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.
- [87] Kensuke Kitai, Yuki Nishida, and Yoshihide Watanabe. “Max-plus Algebraic Description of Evolutions of Weighted Timed Event Graphs”. In: *Theory of Computing Systems* 68.5 (2024), pp. 1427–1441. DOI: **10.1007/s00224-024-10191-x**. URL: **<https://doi.org/10.1007/s00224-024-10191-x>**.
- [88] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. “Static analysis of communication for asynchronous concurrent programming languages”. In: *Static Analysis*. Ed. by Alan Mycroft. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 225–242. ISBN: 978-3-540-45050-4.
- [89] Jan Komenda et al. “Max-plus algebra in the history of discrete event systems”. In: *Annual Reviews in Control* 45 (2018), pp. 240–249. DOI: **10.1016/j.arcontrol.2018.04.004**. URL: **<https://doi.org/10.1016/j.arcontrol.2018.04.004>**.
- [90] Jonathan Kotker, Dorsa Sadigh, and Sanjit A. Seshia. “Timing analysis of interrupt-driven programs under context bounds”. In: *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2011, pp. 81–90.
- [91] Ron Koymans. “Specifying real-time properties with metric temporal logic”. In: *Real-time systems* 2.4 (1990), pp. 255–299.
- [92] Daniel Kroening and Ofer Strichman. “Efficient computation of recurrence diameters”. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2003, pp. 298–309.
- [93] Daniel Kroening and Michael Tautschnig. “CBMC–C bounded model checker”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2014, pp. 389–391.
- [94] Daniel Kroening et al. “Linear Completeness Thresholds for Bounded Model Checking”. In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV ’11)* (2011), pp. 557–572.

- [95] Logan Kugler. *How NASA Built Artemis II's Fault-Tolerant Computer*. <https://cacm.acm.org/news/how-nasa-built-artemis-iis-fault-tolerant-computer/>. Communications of the ACM, News. Accessed: 2026-04-10. Apr. 2026.
- [96] Frederick Kuhl, Richard Weatherly, and Judith Dahmann. *Creating Computer Simulation Systems: an Introduction to the High Level Architecture*. Prentice Hall PTR, 1999.
- [97] R. L. Graham. “Bounds on Multiprocessing Timing Anomalies”. In: *SIAM Journal on Applied Mathematics* 17.2 (1969), pp. 416–429.
- [98] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (1978), 558–565. ISSN: 0001-0782. DOI: [10.1145/359545.359563](https://doi.org/10.1145/359545.359563). URL: <https://doi.org/10.1145/359545.359563>.
- [99] Chris Lattner and Vikram S. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2004)*. IEEE Computer Society, 2004, pp. 75–88. DOI: [10.1109/CGO.2004.1281665](https://doi.org/10.1109/CGO.2004.1281665). URL: <https://doi.org/10.1109/CGO.2004.1281665>.
- [100] Steven Lauterburg et al. “A framework for state-space exploration of Java-based actor programs”. In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2009, pp. 468–479.
- [101] E.A. Lee. “The problem with threads”. In: *Computer* 39.5 (2006), pp. 33–42. DOI: [10.1109/MC.2006.180](https://doi.org/10.1109/MC.2006.180).
- [102] Edward Lee and David Messerschmitt. “Synchronous Data Flow”. In: 75.9 (1987), pp. 1235–1245.
- [103] Edward A Lee. “Cyber physical systems: Design challenges”. In: *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*. IEEE. 2008, pp. 363–369.
- [104] Edward A. Lee. “Determinism”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 20.5 (2021), pp. 1–34. DOI: [10.1145/3453652](https://doi.org/10.1145/3453652).
- [105] Edward A Lee. “Recurrences, iteration, and conditionals in statically scheduled block diagram languages”. In: *VLSI Signal Processing* 3 (1988), pp. 330–340.
- [106] Edward A. Lee and David G. Messerschmitt. “Synchronous data flow”. In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245. DOI: [10.1109/PROC.1987.13876](https://doi.org/10.1109/PROC.1987.13876).
- [107] Edward A. Lee and Marjan Sirjani. “What Good are Models?” In: *Formal Aspects of Component Software (FACS)*. Vol. LNCS 11222. Springer, 2018.
- [108] Edward A. Lee et al. “Consistency vs. Availability in Distributed Cyber-Physical Systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* (2023). Presented at EMSOFT, September 17-22, 2023, Hamburg, Germany. DOI: [10.1145/3609119](https://doi.org/10.1145/3609119).

- [109] Edward A. Lee et al. “Consistency vs. Availability in Distributed Cyber-Physical Systems”. In: *ACM Trans. Embed. Comput. Syst.* 22.5s (Sept. 2023). ISSN: 1539-9087. DOI: **10.1145/3609119**. URL: <https://doi.org/10.1145/3609119>.
- [110] Edward A. Lee et al. “Discrete-Event Models”. In: *System Design, Modeling, and Simulation using Ptolemy II*. Ed. by Claudius Ptolemaeus. Berkeley, CA: Ptolemy.org, 2014. ISBN: 978-1-304-42106-7. URL: <http://ptolemy.org/books/Systems>.
- [111] Edward A. Lee et al. “Quantifying and Generalizing the CAP Theorem”. In: *arXiv:2109.07771 [cs.DC]* (2021). URL: <https://arxiv.org/abs/2109.07771>.
- [112] Edward A. Lee et al. “Quantifying and Generalizing the CAP Theorem”. In: *CoRR* abs/2109.07771 (2021). arXiv: **2109.07771**. URL: <https://arxiv.org/abs/2109.07771>.
- [113] Edward A. Lee et al. “Trading Off Consistency and Availability in Tiered Heterogeneous Distributed Systems”. In: *Intelligent Computing 2* (2023), p. 0013. DOI: **10.34133/icomputing.0013**. eprint: <https://spj.science.org/doi/pdf/10.34133/icomputing.0013>. URL: <https://spj.science.org/doi/abs/10.34133/icomputing.0013>.
- [114] Edward Ashford Lee. *Plato and the Nerd — The Creative Partnership of Humans and Technology*. MIT Press, 2017.
- [115] Edward Ashford Lee and David G. Messerschmitt. “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing”. In: *IEEE Transactions on Computers* C-36.1 (1987), pp. 24–35. DOI: **10.1109/TC.1987.5009446**.
- [116] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.
- [117] Wenchao Li. “Specification Mining: New Formalisms, Algorithms and Applications”. PhD thesis. EECS Department, University of California, Berkeley, 2014. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-20.html>.
- [118] Stan Liao, Steve Tjiang, and Rajesh Gupta. “An efficient implementation of reactivity for modeling hardware in the Scenic design environment”. In: *Proceedings of the 34th annual Design Automation Conference*. 1997, pp. 70–75.
- [119] Ben Lickly et al. “Predictable Programming on a Precision Timed Architecture”. In: *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES ’08. Atlanta, GA, USA: Association for Computing Machinery, 2008, 137–146. ISBN: 9781605584690. DOI: **10.1145/1450095.1450117**. URL: <https://doi.org/10.1145/1450095.1450117>.

- [120] Shaokai Lin and Edward A. Lee. “Verify Engineering Models, Not Scientific Models”. In: *Rebeca for Actor Analysis in Action — Essays Dedicated to Marjan Sirjani on the Occasion of Her 60th Birthday*. Ed. by Edward A. Lee, Mohammad Reza Mousavi, and Carolyn L. Talcott. Lecture Notes in Computer Science. Springer, 2025, pp. 195–211. DOI: [10.1007/978-3-031-85134-6_9](https://doi.org/10.1007/978-3-031-85134-6_9). URL: https://doi.org/10.1007/978-3-031-85134-6_9.
- [121] Shaokai Lin et al. *PretVM: Predictable, Efficient Virtual Machine for Real-Time Concurrency*. 2024. arXiv: [2406.06253](https://arxiv.org/abs/2406.06253) [eess.SY]. URL: <https://arxiv.org/abs/2406.06253>.
- [122] Shaokai Lin et al. “Towards Building Verifiable CPS Using Lingua Franca”. In: *ACM Trans. Embed. Comput. Syst.* 22.5s (2023). ISSN: 1539-9087. DOI: [10.1145/3609134](https://doi.org/10.1145/3609134). URL: <https://doi.org/10.1145/3609134>.
- [123] Marten Lohstroh et al. “A Language for Deterministic Coordination Across Multiple Timelines”. In: *Forum for Specification and Design Languages (FDL)*. IEEE, 2020. DOI: [10.1109/FDL50818.2020.9232939](https://doi.org/10.1109/FDL50818.2020.9232939).
- [124] Marten Lohstroh et al. “A language for deterministic coordination across multiple timelines”. In: *2020 Forum for Specification and Design Languages (FDL)*. IEEE, 2020, pp. 1–8.
- [125] Marten Lohstroh et al. “Logical Time for Reactive Software”. In: *Workshop on Timing-Centric Reactive Software (TCRS), in Cyber-Physical Systems and Internet of Things Week (CPSIoT)*. ACM, 2023. DOI: [10.1145/3576914.3587494](https://doi.org/10.1145/3576914.3587494).
- [126] Marten Lohstroh et al. “Reactors: A deterministic model for composable reactive systems”. In: *Model-Based Design of Cyber Physical Systems (CyPhy’19)* (2019).
- [127] Marten Lohstroh et al. “Reactors: A Deterministic Model for Composable Reactive Systems”. In: *8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy’19)*. Vol. LNCS 11971. in press. Springer-Verlag, 2019.
- [128] Marten Lohstroh et al. “Toward a Lingua Franca for Deterministic Concurrent Systems”. In: *ACM Trans. Embed. Comput. Syst.* 20.4 (2021). ISSN: 1539-9087. DOI: [10.1145/3448128](https://doi.org/10.1145/3448128). URL: <https://doi.org/10.1145/3448128>.
- [129] Marten Lohstroh et al. “Toward a Lingua Franca for Deterministic Concurrent Systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 20.4 (2021), Article 36. DOI: [10.1145/3448128](https://doi.org/10.1145/3448128).
- [130] Marten Lohstroh et al. “Toward a Lingua Franca for Deterministic Concurrent Systems”. In: *ACM Trans. Embed. Comput. Syst.* 20.4 (2021). ISSN: 1539-9087.
- [131] Thomas Lundqvist and Per Stenström. “Timing Anomalies in Dynamically Scheduled Microprocessors”. In: *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 12–21. ISBN: 0-7695-0475-2.

- [132] Daniel Lustig et al. “COATCheck: Verifying Memory Ordering at the Hardware-OS Interface”. In: *SIGARCH Comput. Archit. News* 44.2 (2016), 233–247. ISSN: 0163-5964. DOI: [10.1145/2980024.2872399](https://doi.org/10.1145/2980024.2872399). URL: <https://doi.org/10.1145/2980024.2872399>.
- [133] Claire Maiza et al. “A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems”. In: *ACM Comput. Surv.* 52.3 (June 2019). ISSN: 0360-0300. DOI: [10.1145/3323212](https://doi.org/10.1145/3323212). URL: <https://doi.org/10.1145/3323212>.
- [134] Oded Maler, Zohar Manna, and Amir Pnueli. “From Timed to Hybrid Systems”. In: *Real-Time: Theory and Practice, REX Workshop*. Springer-Verlag, 1992, pp. 447–484.
- [135] Zohar Manna and Amir Pnueli. “Verifying Hybrid Systems”. In: *Hybrid Systems*. Vol. LNCS 736. 1993, pp. 4–35.
- [136] Mircea Marin et al. “Semantics and Formal Analysis of Lingua Franca CPS Specifications in Rewriting Logic”. In: *Rebeca for Actor Analysis in Action: Essays Dedicated to Marjan Sirjani on the Occasion of Her 60th Birthday*. Ed. by Edward A. Lee, Mohammad Reza Mousavi, and Carolyn Talcott. Cham: Springer Nature Switzerland, 2025, pp. 70–101. ISBN: 978-3-031-85134-6. DOI: [10.1007/978-3-031-85134-6_4](https://doi.org/10.1007/978-3-031-85134-6_4). URL: https://doi.org/10.1007/978-3-031-85134-6_4.
- [137] Emad Jacob Maroun et al. “The Platin Multi-Target Worst-Case Analysis Tool”. In: *22th International Workshop on Worst-Case Execution Time Analysis (WCET 2024)*. 2024.
- [138] Christian Menard et al. “Achieving Determinism in Adaptive AUTOSAR”. In: *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE)*. DATE ’20. Grenoble, France: EDA Consortium, Mar. 2020.
- [139] Christian Menard et al. “High-performance Deterministic Concurrency Using Lingua Franca”. In: *ACM Trans. Archit. Code Optim.* 20.4 (2023). ISSN: 1544-3566. DOI: [10.1145/3617687](https://doi.org/10.1145/3617687). URL: <https://doi.org/10.1145/3617687>.
- [140] Christian Menard et al. “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 Proceedings*. DroneSE and RAPIDO ’21. Budapest, Hungary: Association for Computing Machinery, 2021, 66–73. ISBN: 9781450389525. DOI: [10.1145/3444950.3447285](https://doi.org/10.1145/3444950.3447285). URL: <https://doi.org/10.1145/3444950.3447285>.
- [141] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.
- [142] Aleksandar Milicevic and Hillel Kugler. “Model checking using SMT and theory of lists”. In: *NASA Formal Methods Symposium (NFM)*. Springer. 2011, pp. 282–297.
- [143] Walid A. Najjar, Edward A. Lee, and Guang R. Gao. “Advances in the dataflow computational model”. In: *Parallel Computing* 25.13-14 (1999), pp. 1907–1929.

- [144] NASA. *Artemis II: NASA's First Crewed Lunar Flyby in 50 Years*. <https://www.nasa.gov/mission/artemis-ii/>. Accessed: 2026-04-10. 2026.
- [145] George C. Necula et al. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs". In: *Compiler Construction, 11th International Conference (CC 2002), held as part of ETAPS 2002*. Ed. by R. Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 213–228. DOI: **10.1007/3-540-45937-5_16**. URL: https://doi.org/10.1007/3-540-45937-5_16.
- [146] Joël Ouaknine and James Worrell. "On the decidability and complexity of metric temporal logic over finite words". In: *arXiv preprint cs/0702120* (2007).
- [147] Joel Ouaknine and James Worrell. "On the decidability and complexity of Metric Temporal Logic over finite words". In: *Logical Methods in Computer Science* 3.1 (2007). Ed. by Prakash Panangaden. DOI: **10.2168/lmcs-3(1:8)2007**.
- [148] Joël Ouaknine and James Worrell. "Safety Metric Temporal Logic Is Fully Decidable". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Holger Hermanns and Jens Palsberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 411–425. ISBN: 978-3-540-33057-8.
- [149] Joël Ouaknine and James Worrell. "Some recent results in metric temporal logic". In: *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer. 2008, pp. 1–13.
- [150] Francesco Paladino, Shulu Li, and Edward A. Lee. *Maxwait: A Generalized Mechanism for Distributed Time-Sensitive Systems*. 2026. arXiv: **2601.21146 [cs.DC]**. URL: <https://arxiv.org/abs/2601.21146>.
- [151] Terence Parr. "The definitive ANTLR 4 reference". In: *The Definitive ANTLR 4 Reference* (2013), pp. 1–326.
- [152] Elizabeth Polgreen et al. "UCLID5: Multi-modal formal modeling, verification, and synthesis". In: *International Conference on Computer Aided Verification*. Springer. 2022, pp. 538–551.
- [153] Matteo Pradella, Angelo Morzenti, and Pierluigi San Pietro. "Bounded Satisfiability Checking of Metric Temporal Logic Specifications". In: *ACM Trans. Softw. Eng. Methodol.* 22.3 (2013). ISSN: 1049-331X. DOI: **10.1145/2491509.2491514**. URL: <https://doi.org/10.1145/2491509.2491514>.
- [154] Claudius Ptolemaeus, ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.
- [155] Muntasir Raihan Rahman et al. "Characterizing and Adapting the Consistency-Latency Tradeoff in Distributed Key-Value Stores". In: *ACM Transactions on Autonomous and Adaptive Systems* 11.4 (2017), Article 20. DOI: **10.1145/2997654**.
- [156] Ragunathan Rajkumar et al. "Cyber-physical systems: the next computing revolution". In: *Design automation conference*. IEEE. 2010, pp. 731–736.

- [157] Arni Hermann Reynisson et al. “Modelling and simulation of asynchronous real-time systems using Timed Rebeca”. In: *Science of Computer Programming* 89 (2014), pp. 41–68. DOI: [10.1016/j.scico.2014.01.008](https://doi.org/10.1016/j.scico.2014.01.008).
- [158] Arni Hermann Reynisson et al. “Modelling and simulation of asynchronous real-time systems using Timed Rebeca”. In: *Science of Computer Programming* 89 (2014), pp. 41–68.
- [159] Marcus Rossel et al. “Provable Determinism for Software in Cyber-Physical Systems”. In: *Verified Software. Theories, Tools and Experiments*. Ed. by Andrew Reynolds and Serdar Tasiran. Cham: Springer Nature Switzerland, 2024, pp. 85–107. ISBN: 978-3-031-66064-1.
- [160] Guillaume Roumage et al. “An Extended Survey and a Comparison Framework for Dataflow Models of Computation and Communication”. In: *arXiv preprint arXiv:2501.07273* (2025).
- [161] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. “Effpi: verified message-passing programs in Dotty”. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*. 2019, pp. 27–31.
- [162] Martin Schoeberl et al. “Invited Paper: Worst-Case Execution Time Analysis of Lingua Franca Applications”. English. In: vol. 121. 22nd International Workshop on Worst-Case Execution Time Analysis, WCET ; Conference date: 09-07-2024 Through 12-07-2024. Schloss Dagstuhl- Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, 2024. DOI: [10.4230/OASIcs.WCET.2024.4](https://doi.org/10.4230/OASIcs.WCET.2024.4).
- [163] Martin Schoeberl et al. “Patmos: A time-predictable microprocessor”. In: *Real-Time Systems* 54.2 (2018), pp. 389–423.
- [164] Alexander Schulz-Rosengarten et al. “Modal Reactors”. In: *arXiv preprint arXiv:2301.09597* (2023).
- [165] Bart De Schutter et al. “Analysis and Control of Max-Plus Linear Discrete-Event Systems: An Introduction”. In: *Discrete Event Dynamic Systems* 30.1 (2020), pp. 25–54. DOI: [10.1007/s10626-019-00294-w](https://doi.org/10.1007/s10626-019-00294-w). URL: <https://doi.org/10.1007/s10626-019-00294-w>.
- [166] Reinhard Schwarz and Friedemann Mattern. “Detecting causal relationships in distributed computations: In search of the holy grail”. In: *Distributed computing* 7.3 (1994), pp. 149–174.
- [167] Sanjit A. Seshia. “Combining Induction, Deduction, and Structure for Verification and Synthesis”. In: *Proceedings of the IEEE* 103.11 (2015), pp. 2036–2051.
- [168] Sanjit A. Seshia and Jonathan Kotker. “GameTime: A Toolkit for Timing Analysis of Software”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Parosh Aziz Abdulla and K. Rustan M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 388–392. ISBN: 978-3-642-19835-9. URL: <https://berkeleylearnverify.github.io/GameTime/>.

- [169] Sanjit A. Seshia and Alexander Rakhlin. “Game-theoretic Timing Analysis”. In: *2008 International Conference on Computer-Aided Design (ICCAD 2008)*. IEEE Computer Society, 2008, pp. 575–582. DOI: [10.1109/ICCAD.2008.4681634](https://doi.org/10.1109/ICCAD.2008.4681634). URL: <https://doi.org/10.1109/ICCAD.2008.4681634>.
- [170] Sanjit A. Seshia and Alexander Rakhlin. “Quantitative Analysis of Systems Using Game-Theoretic Learning”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 11.S2 (2012), 55:1–55:27.
- [171] Sanjit A. Seshia and Pramod Subramanyan. “UCLID5: Integrating Modeling, Verification, Synthesis, and Learning”. In: *Proceedings of the 15th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. 2018.
- [172] Gilbert C. Sih. “Multiprocessor Scheduling to Account for Interprocessor Communication”. PhD thesis. EECS Department, University of California, Berkeley, 1991. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/1991/1729.html>.
- [173] Marjan Sirjani and Mohammad Mahdi Jaghoor. “Ten Years of Analyzing Actors: Rebeca Experience”. In: *Formal Modeling: Actors, Open Systems, Biological Systems*. Ed. by Agha G., Danvy O., and Meseguer J. Vol. Lecture Notes in Computer Science, vol 7000. Berlin, Heidelberg: Springer, 2011. DOI: [10.1007/978-3-642-24933-4_3](https://doi.org/10.1007/978-3-642-24933-4_3).
- [174] Marjan Sirjani and Ehsan Khamespanah. “On Time Actors”. In: *Theory and Practice of Formal Methods* LNCS 9660 (2016). Springer, Cham, pp. 373–392. DOI: [10.1007/978-3-319-30734-3_25](https://doi.org/10.1007/978-3-319-30734-3_25).
- [175] Marjan Sirjani, Edward A. Lee, and Ehsan Khamespanah. “Verification of Cyber-physical Systems”. In: *Mathematics* 8 (2020), pp. 1–20. DOI: [10.3390/math8071068](https://doi.org/10.3390/math8071068).
- [176] Marjan Sirjani, Edward A. Lee, and Ehsan Khamespanah. “Verification of Cyber-physical Systems”. In: *Mathematics* 8.7 (2020). ISSN: 2227-7390. DOI: [10.3390/math8071068](https://doi.org/10.3390/math8071068). URL: <https://www.mdpi.com/2227-7390/8/7/1068>.
- [177] Sander Stuijk et al. “Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications”. In: *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. 2011, pp. 404–411. DOI: [10.1109/SAMOS.2011.6045491](https://doi.org/10.1109/SAMOS.2011.6045491).
- [178] Pramod Subramanyan and Sanjit A Seshia. *Getting started with Uclid5*. 2021.
- [179] Binqi Sun et al. “Edge Generation Scheduling for DAG Tasks Using Deep Reinforcement Learning”. In: *IEEE Transactions on Computers* 73.4 (2024), pp. 1034–1047.
- [180] **Shaokai Lin** et al. “Navigating Time and Energy Tradeoffs in Reactive Heterogeneous Systems”. In: *IEEE Embedded Systems Letters* 17.2 (2025), pp. 103–106. DOI: [10.1109/LES.2024.3469278](https://doi.org/10.1109/LES.2024.3469278).

- [181] **Shaokai Lin** et al. “Quasi-Static Scheduling for Deterministic Timed Concurrent Models on Multi-Core Hardware”. In: *ACM Trans. Embed. Comput. Syst.* 24.5s (Sept. 2025). ISSN: 1539-9087. DOI: **10.1145/3762653**. URL: **<https://doi.org/10.1145/3762653>**.
- [182] *The industry standard for static timing analysis*. URL: **<https://www.absint.com/ait/index.htm>**.
- [183] L. Thiele, S. Chakraborty, and M. Naedele. “Real-time calculus for scheduling hard real-time systems”. In: *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*. Vol. 4. 2000, 101–104 vol.4. DOI: **10.1109/ISCAS.2000.858698**.
- [184] UCLID5 contributors. *UCLID5: Formal modeling, verification, and synthesis of computational systems*. **<https://github.com/uclid-org/uclid>**. GitHub repository. Accessed 2026-05-14.
- [185] Micaela Verucchi, Ignacio Sañudo Olmedo, and Marko Bertogna. “A survey on real-time dag scheduling, revisiting the global-partitioned infinity war”. In: *Real-Time Systems* 59.3 (2023), pp. 479–530.
- [186] Micaela Verucchi et al. “Latency-aware generation of single-rate DAGs from multi-rate task sets”. In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020, pp. 226–238.
- [187] Zachariah Wasson. “Analyzing Data-Dependent Timing and Timing Repeatability with GameTime”. MA thesis. EECS Department, University of California, Berkeley, May 2014. URL: **<https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-132.html>**.
- [188] Andrew Waterman et al. “The RISC-V instruction set manual”. In: *Volume I: User-Level ISA’, version 2* (2014).
- [189] Jonatan Wiik and Pontus Boström. “Specification and automated verification of dynamic dataflow networks”. In: *International Conference on Software Engineering and Formal Methods*. Springer. 2017, pp. 136–151.
- [190] Xronos. *Build Intelligent Systems with Xronos*. **<https://www.xronos.com>**. From prototype to production, ready-to-deploy software infrastructure powers scalable and safe AI-driven robotics and mobility solutions. Accessed: 2026-05-11. n.d.
- [191] Shohei Yasutake and Takuo Watanabe. “Actario: A framework for reasoning about actor systems”. In: *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE)*. 2015.
- [192] Haifeng Yu and Amin Vahdat. “Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services”. In: *ACM Transactions on Computer Systems* 20.3 (2002), pp. 239–282. DOI: **10.1145/566340.566342**. URL: **<https://doi.org/10.1145/566340.566342>**.

- [193] Haifeng Yu and Amin Vahdat. “The costs and limits of availability for replicated services”. In: *ACM Transactions on Computer Systems* 2.24 (2006), pp. 70–113. DOI: **10.1145/1124153.1124156**.
- [194] Bernard P Zeigler et al. “The DEVS environment for high-performance modeling and simulation”. In: *IEEE Computational Science and Engineering* 4.3 (1997), pp. 61–71.
- [195] Yang Zhao, Edward A. Lee, and Jie Liu. “A Programming Model for Time-Synchronized Distributed Real-Time Systems”. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2007, pp. 259–268.
- [196] Adam Ziebinski et al. “Review of advanced driver assistance systems (ADAS)”. In: *AIP Conference Proceedings* 1906.1 (2017), p. 120002. DOI: **10.1063/1.5012394**. eprint: <https://aip.scitation.org/doi/pdf/10.1063/1.5012394>. URL: <https://aip.scitation.org/doi/abs/10.1063/1.5012394>.
- [197] Michael Zimmer et al. “FlexPRET: A processor platform for mixed-criticality systems”. In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2014, pp. 101–110. DOI: **10.1109/RTAS.2014.6925994**.