Modelling real world applications in Lingua Franca

Robin Mithoff

Bachelorthesis
September 28, 2023

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
Malte Clement

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Abstract

In contemporary software development, creating concurrent, distributed, and real-time systems is crucial, especially for industrial companies like Scheidt & Bachmann, which focus on safety-critical transport and railway systems. The conventional computational models, including SCCharts, employed for designing these systems often compromise determinism. The current thesis investigates the applicability and benefits of utilizing Lingua Franca, a coordination language, as a supplementary tool for documenting and testing such systems developed at Scheidt & Bachmann using SCCharts. Through a careful exploration and re-implementation of certain SCCharts-modeled components into Lingua Franca, the study sought to understand Lingua Franca's advantages, features, and constraints within a real-world system context. Two approaches were primarily explored - a conceptual rebuilding in Lingua Franca and orchestrating interaction between components in Lingua Franca. Subsequent analysis identified that while direct re-implementation might introduce complexities, particularly in capturing state transitions and concurrent behaviors, one approach, using Modal Reactor within Lingua Franca revealed a more consistent and intuitive pattern for modeling and documenting concurrent interactions. The study, however, concluded that while Lingua Franca presented a coherent and intuitive modeling alternative with deterministic behavior assurance, the task of rebuilding existing SCChart systems solely for enhanced documentation was deemed infeasible for Scheidt & Bachmann. The thesis suggests a potential direction for future work in integrating SCCharts within Lingua Franca to leverage event-driven behaviors inherent in both paradigms, aiming towards a more robust, testable, and well-documented concurrent distributed system. The insights obtained from this investigation lay out a foundation for exploring an integrated approach that might foster better testability and reliability in concurrent distributed systems' development and maintenance.

Acknowledgements

Embarking on this thesis journey has been an enlightening experience, revealing the intricacies of the complex topics I aimed to explore. My time spent working alongside the committed teams at Kiel University and Scheidt & Bachmann has not only propelled my knowledge and understanding to new heights but has also left a lasting impression on my academic trajectory.

I extend my sincere appreciation to Prof. Dr. Reinhard von Hanxleden for his unwavering support and keen insights throughout this endeavor. His encouragement and straightforwardness have been crucial in navigating the challenges encountered along the way. I am indebted to him for providing a conducive work environment and for his genuine care towards my academic progression.

My gratitude also goes to Malte Clement, whose patience and dedicated supervision have been nothing short of inspiring. His availability and constructive advice have been instrumental in refining my understanding and execution of the task at hand. The journey would have been a lot tougher without his encouraging words and valuable feedback.

I am grateful for the technical guidance and insights shared by Alexander Schulz-Rosengarten. His profound knowledge of Lingua Franca considerably enriched my work, aiding in overcoming technical hurdles that seemed insurmountable.

The practical experience and professional relationship fostered with Scheidt & Bachmann have significantly contributed to my thesis. Their technical acumen, supportive work environment, and the real-world perspectives they offered have been invaluable. The financial support provided also eased the journey, for which I am thankful.

A special note of thanks goes to Hauke Fuhrmann, Deputy Head of Development, for making it possible to undertake my thesis at Scheidt & Bachmann. His patience, timely advice, and unwavering support despite his tight schedule, are highly appreciated.

Nis Wechselberg and Merlin Kötzing have been remarkable in providing the necessary technical knowledge regarding the systems at Scheidt & Bachmann, which were integral to my project. Their openness and readiness to assist are attributes that have made a positive impact on my work.

Further, I appreciate the effort of Nis Wechselberg, Merlin Kötzing, Daniel Grevismühl, and Malte Clement in participating in my survey, contributing critical data that was essential for the analyses performed.

This thesis marks the culmination of an enriching phase of my academic journey at Kiel University. The knowledge gained, relationships formed, and experiences shared have been formative, leaving a significant imprint as I transition into the next phase of my life.

Lastly, to my peers, friends, and family who have been a source of moral support, thank you for being there, for the discussions that sparked insight, and for the shared moments that lightened the journey. Your belief in me has been a pillar of strength, propelling me towards the finish line with enthusiasm and resolve.

Contents

List of Figures

List of Tables

List of Tables

PTIDES	Programming Temporally Integrated Distributed Embedded Systems
RTI	Run Time Infrastructure
LF	Lingua Franca Coordination Language
iUZ	*IntegrierteUnterzentrale*
SCChart	Sequentially Constructive Chart
S&B	*Scheidt&Bachmann*
iBS	*IntegriertesBediensystem*
iBS-Z	*integriertesBediensystem – Zentrale*
iBP	*integrierterBedienplatz*
RS	railway systems
ADAS	advanced driver-assistance systems
OS	Operating Station

Chapter 1

Introduction

One of the many interesting tasks in software development is to build concurrent, distributed, time sensitive and reactive systems. Contemporary computational models used to build concurrent systems, such as Publish-Subscribe frameworks, actor models or shared memory concepts, often sacrifice determinism [MLB+23].

Publish-Subscribe is a messaging protocol used in concurrent and distributed systems where the publishers send messages without the need to know who will receive them. The subscribers express interest in one or more messages, and only receive messages that are of interest, without the need to know who sent them. Messages are typically sent to a central broker or event bus that manages the distribution, even though it could be achieved without a central intermediary by invoking events in the subscriber directly. The Publish-Subscribe model often involves asynchronous communication, where the publisher does not wait for a response from the subscriber, hence there is no ordering guaranty of the messages by default. This asynchronous behavior can lead to non-determinism because the order in which messages are received and processed by subscribers can vary based on timing, network delays, and system load. Additionally, if multiple subscribers are competing for resources, the order of message processing can also vary, leading to different outcomes in different runs.

Actors, briefly discussed in Chapter 2, are mathematical models that are used for concurrent computation. In the actor model, actors are the primary units of computation, each encapsulating state and behavior, that can send and receive messages and have an internal changeable state. Actors only process one message at a time and communicate exclusively through message passing, ensuring isolation. Actors reference each other and therefore share the internal information with other connected actors. In a distributed system, when using the TCP/IP-Protocol, message ordering is guarantied by definition for the communication between two actors.

With an actor receiving two messages, each from a different actor, there is no guarantee in general in which order those messages are processed, thus sacrificing deterministic behavior.

Shared memory protocols, and implementations of concurrent systems using that protocol, often use multiple threads, reading and writing to a shared memory space.

If not handled with caution, race conditions can occur if two or more threads access shared data at the same time. Even when basic solutions such as shared memory locks are used, other problems such as deadlocks can occur. Therefore the outcome of a system using shared memory depends on many factors, which subsequently leads to non-deterministic behavior. However, even if there is a policy in place such as "always use the most recent data", failures can not be ruled out, as the example of NASA's Toyota Study[1] shows. The study involved a detailed analysis of Toyota´s electronic throttle control system that caused at some instances an unintended acceleration of the car. The system involved multiple microprocessors that communicate with each other using shared memory. While there was no electronic flaw found in the system, the study concluded that the system is untestable. This results from the vast number of possible states that are caused by the memory sharing and policy of accessing it. Hence, the absence of determinism in concurrent and distributed systems, can cause failures not only in the system itself, it also makes it difficult to test for unintended behavior.

Therefore the need for reliable concurrent and distributed systems is an important aspect for many industrial companies that build safety-critical real time systems, such as *Scheidt&Bachmann* (S&B), that specialize in transport and railway systems (RS). The *IntegriertesBediensystem* (iBS) is one of their products that is a collection of many integrated components with a specialized purpose. One of those components is the *integrierterBedienplatz* (iBP) that allows RS operators to control, manipulate and communicate with connected components such as interlocking systems, level crossings, cameras and more from a user interface. The logic behind the iBP is controlled by yet another component of the iBS, the *integriertesBediensystem − Zentrale* (iBS-Z). In the process of accessing distributed components there is often another product, the *IntegrierteUnterzentrale* (iUZ), as middleware involved. At present the iBS and the iUZ and its subunits consist of several sub components, which themself provide stand-alone services. Those sub components communicate with each other over a network if necessary in order to provide a service.

These components are currently modeled using a customized version of Sequentially Constructive Charts (SCCharts). SCCharts is a visual modeling language developed at Kiel University. It is designed to specify and implement reactive, real-time systems in a modular and compositional manner. It was developed to overcome several of the limitations and ambiguities associated with the original statechart formalism [HDM+14]. The synchronous model of computation helps, among other features, to ensure deterministic behavior in systems built with SCCharts. The graphical representation of SCCharts aims to make system specifications, behavior and communication,

[1] https://www.nasa.gov/topics/nasalife/features/nesc-toyota-study.html

more intuitive and easier to reason about and therefore potentially be supplemental for documentation. Nonetheless, some of the sub components of S&Bs products are modeled and build in separate SCChart instances, even if they interact with each other in a running system. While this still provides the features of SCCharts for the sub system itself, deterministic behavior between those components can not be guarantied by default. Furthermore, SCChart diagrams might help to understand the semantics of the sub system, the interaction between different components is not represented in the same way. S&B seeks to evaluate alternatives that can be used in addition to the contemporary documentation of systems build with SCCharts. Moreover, writing concurrent and distributed software systems that are verifiable and testable is not a trivial task, due to challenges regarding consistency and availability in these systems, even for professional developers at S&B.

This thesis explores the Lingua Franca Coordination Language (LF) as a tool for additional documentation of systems build at S&B with SCChart. In addition to that, LF is evaluated as a potential tool for testing or simulating systems in terms of the communication of concurrency. LF is a coordination language designed to address the challenges associated with concurrent and distributed systems, particularly those that require precise timing and deterministic behavior [MLB+23]. It is not a programming language by itself. Programs written in LF are processed by the LF-Codegenerator that outputs target code for the specified target language. LF is based on a reactive model of computation, where the system reacts to events that occur at discrete points in time. The timing of events is precisely defined, and the reactions to events are deterministic. It is possible to build entire systems from scratch in LF, because the main functionality is derived from the specified target language. Nonetheless, the focus lies on the orchestration of the interaction between components and guaranteeing deterministic behavior and precise timing on concurrent and distributed systems. In order to achieve the desired conditions LF utilizes the concept of logical time and, specific scheduling policies and the Reactor model is a deterministic extension of the actor model [LL19]. Moreover, LF offers interactive diagrams that facilitate navigation through components and their underlying logic. Users can click on reactors and reactions to expand or collapse their details.

1.1 Related Work

The author is not aware of any other work done that specifically aimed at rebuilding real world systems in LF using SCChart notation. Although, many publications and contributions by the LF community have been instrumental and crucial for this thesis. They provided practical and real world examples, that which influenced the

understanding and application of LF while developing different systems throughout this this endeavor.

In the domain of distributed embedded systems, especially concerning safety-critical applications like autonomous driving, the issues of nondeterminism arising from existing asynchronous frameworks are prominent. An attempt to address this problem is demonstrated in the work by [BLW+22], where the authors introduced Xronos, an open-source framework built on top of Lingua LF. LF, akin to SCChart, incorporates the reactor model facilitating deterministic interactions among various physical and logical timelines, therefore advancing the predictability of system behavior. The discussed case study centers around transitioning the open-source autonomous driving software, Autoware.Auto[2], from ROS[3] to Xronos. This transition showcases notable advancements in addressing nondeterminism, identifying timing faults, and enhancing the predictability of coordination across components. This endeavor reflects the core spirit of the translational efforts explored in this thesis, albeit through a different notation and within a distinct domain.

The paper by [LBL+23a] explores the challenges in tiered distributed computing systems, particularly focusing on data consistency and availability amidst varying network latencies. The authors introduce the Consistency, Availability, apparent Latency (CAL) theorem and apply LF to manage these challenges. Through LF, they explicitly define availability and consistency requirements, and utilize the CAL theorem to derive essential network latency bounds, guiding the system design within these heterogeneous network environments. This work is relevant to this thesis as it showcases a practical application of LF in addressing real-world system design challenges, and it provides a precedent for the exploration carried out in this thesis regarding the use of LF and SCChart notation in rebuilding real-world systems.

The PhD thesis by [Loh20] provides a number of practical examples that were significantly helpful in understanding different features and aspects of LF and in developing systems for the present work.

1.2 Problem Statement

To investigate potential benefits of leveraging LF for the development process at S&B,it is necessary to explore how system components, modeled with SCCharts, can be translated or re-implemented into a LF system. This will shed light on the advantages, features and limitations of LF within the context of an existing real world system.

[2]https://autoware.org/
[3]https://www.ros.org/

As previously mentioned, one problem is that if components that interact with each other are modeled separately with SCCharts, the actual interaction might not be obvious from the visualized documentation. Especially in complex distributed systems it potentially adds an auxiliary level for documentation, that is more readily comprehensible.

Currently, S&B uses custom algorithms and tests to make assumptions about the properties of software systems, its components and the networks on which those systems run. This is by no means a trivial and easy to replicate task. While LF offers an alternative approach to build entire systems in a deterministic and testable way, it is not the goal of S&B to find a substitute framework and to rebuild their existing systems with it for several reasons. Instead, they seek an approach that can help orchestrate existing or newly build systems in a way that can be tested even more reliably.

There are two possible ways to approach this: First the targeted system is built in LF on a conceptional level. This means that the functionality that is not required to test the communication of the system components does not need to be implemented. Mocked data can be used to simulate the future functionality. The other approach could be wrapping existing components in Reactors and only orchestrating the interaction between those wrapped components. Both strategies then can be used to test the systems properties more reliably since LF ensures deterministic behavior, regardless of the system being distributed or not. If the system works as required in the orchestrated version using LF, it becomes possible to derive assumptions inherent in the LF model and from there the requirements needed for the actual implementation of the system without LF. Essentially this means mapping the properties observed in the LF version to the actual implementation and thereby improving reliability and testability.

For that goal the features LF offers, especially for distributed execution, may be beneficial. Within LFs distributed execution functionality, it is possible to explicitly trade off availability, consistency and network latency in the existing system [LBL+23b]. This offers flexibility choosing and evaluating the requirements and assumptions of the system.

1.3 Outline

The second chapter introduces the basic functionality and use of LF along with useful features. Furthermore, some of the theoretical concepts that LF is built on and are advantageous to solve the problems mentioned in the problem statement are also discussed in the second chapter. The third chapter contains the implementation of

two system components in LF, that were developed by S&B using SCChart. For that, those sub components, one of the iBS and one iUZ, that are separately built using SCCharts but interact with each other in the actual system, are rebuilt in LF using different strategies. In this isolated state, not all the functionality of the components is modeled, but only the necessary functionality and simulated behavior. The fourth chapter evaluates the implementation of the chapter before and LF as additional layer of documentation. The focus lies on how the system was built in LF explicitly using SCCharts documentation. Moreover, it is assessed how the CAL theorem in combination with LF can be used to derive properties about a system and the communication of distributed units to draw conclusions and formulate requirements based on that conclusions.

Chapter 2

Foundations & Used Technologies

Reactive systems can be defined as systems that are responsive, , elastic and message-driven. Responsiveness means that systems guarantee timely reactions with an emphasis on consistent response times to ensure a uniform quality of service. Resilience means that systems remain operational even during failures. Thus the failure of a component of a system does not imply the entire system fails. Achieving elasticity results in systems that are able to dynamically adjust to workload changes and therefore eliminates central bottlenecks. In the context of LF, asynchronous message-passing is important, because it enables loose coupling and isolation. Furthermore, location-independent messaging provides failure management across diverse environments while non-blocking communication reduces the wastage of resources. A more precise definition of the meaning of reactive systems in modern software development can be found in the "Reactive Manifesto"[1].

One option to model a reactive system with the aforementioned requirements is the Actor-Model introduced in [Hew77], which can be regarded as an evolutionary predecessor of the model used in LF and will be discussed briefly later on. This and other approaches for reactive systems often come with the loss of determinacy as a trade off for the advances in the required properties [LRG+20]. This can result in a decreased level of testability for the system, despite the fact that systematic testing is the prevailing method used to assess the accuracy of software.

LF, as a polyglot coordination language, provides a model that allows for developing reactive systems while maintaining determinism by default. This is achieved through the Reactor-Model with a logical notion of time, an event scheduler and a reactive and synchronous communication. A reactor is a set of routines, called reactions, and share a local state [LRG+20]. There is no need to learn a new programming language in order to use LF, since every program produces real code in the given target language.

The fundamental principles of LF are introduced, where the necessary mathematical formalism is presented to address the particular topics under discussion. A more formal approach, can be found in [LRG+20] and [Loh20]. If not specified differently, the used target language in this thesis is `Python`. It has a rich library support which

[1] https://www.reactivemanifesto.org/

will helps showcasing the integration of external code into a LF system. For the most part of this chapter a very simplified model of a braking system in a car is used for the code examples. The discussed scenarios and resulting design choices may not be realistic in a real world scenario but highlights the features of LF.

LF is still under development, meaning that not all features are accessible for all target languages yet. Furthermore, this thesis does not cover all features of LF or provide instructions on how to install and run it on a specific machine. However, there are numerous publications and tutorial videos available on YouTube, and the official website offers comprehensive documentation. [2].

2.1 Actors

The Actor Model, introduced by Hewitt and Agha [Hew77], presents a high-level computational framework for concurrent and distributed systems. Central to this model are actors, which are autonomous computational entities. Key characteristics and principles include Asynchronous Messaging, Local State, Concurrency and Parallelism, Location Transparency and Fault Tolerance. Actors communicate exclusively through asynchronous messages. On receipt of a message, an actor can execute actions such as sending additional messages, creating new actors, or updating its internal state. Each actor maintains its own private state, which cannot be directly accessed or modified by other actors. This inherent encapsulation offers safe concurrent computations. Every actor processes its messages concurrently, offering a natural way to express parallel computations. The model removes low-level synchronization mechanisms, enabling the creation of scalable systems. In a distributed system, actors can reside on any node in the system. Their addressing is location-independent, facilitating a easier approach for distribution of actors. The Actor Model supports structured ways to handle failures, where actors can supervise the failure of other actors, deciding on recovery strategies.

Imagine a braking system comprised of three components: a unit that processes input from a rear distance sensor, a sensor that detects when the brake is applied, and the physical brake system itself. When the brakes are activated, the system checks for the presence of nearby vehicles that could potentially cause a collision if a full brake is applied at the current velocity. If the distance between the vehicles indicates a collision is imminent, the system adjusts the intensity of the deceleration to the current parameters. In a real-world scenario, it is probably not advisable to adjust the brakes based on such limited data. In 2.1 a message is send to `Sensor` and `Brakes` if the brake pedal is triggered. The `Sensor` then reads the distance and velocity and

[2] www.lf-lang.org

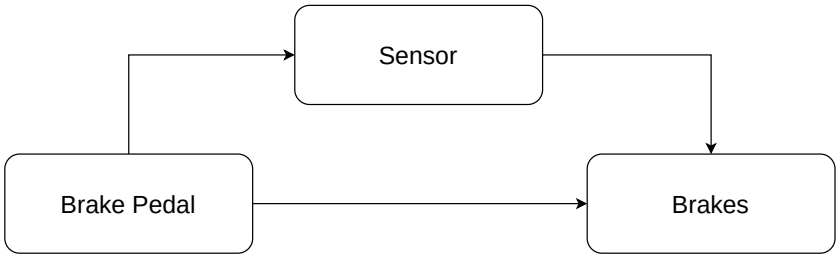

Figure 2.1. Simplified brakeing system model

computes if it is safe to brake under predefined conditions, and forwards the result as a message to the brakes. With this design, after the Brakes component received a message from the brake pedals it arms the brakes and waits as specific amount of time (a considerably small deadline) for the message from the sensor in order to decide if it is safe to decelerate or if moderation is needed. If the deadline is violated the brakes are initiated without moderation. This design assumes that the sensor data for the current brake initiation arrives after the message from the brake Pedal. The question arises: what happens if the messages' arrival does not follow this assumption? If the sensor data bis detected before the braking pedal data, the Brakes component could precisely check for this using a predefined time range in which the messages must be detected to register as a single braking signal. But if the messages arrive at the same logical time, no ordering guarantee for the message processing is given. Thus making this system less testable and therefore not ineligible for production.

The arbitrary issue could potentially be resolved through an alternative design decision. However, this may not be as simple for intricate distributed reactive systems. LF guarantees by design a deterministic ordering of message or event processing even if received at the same logical instance of time.

2.2 Logical Time and Superdense Time

Logical time is a conceptual framework used in distributed systems to order events in a way that reflects their causal relationships, rather than relying on physical time e.g. wall-clock time. Lamport's timestamps [Lam19] serve as one of the foundational methods for representing logical time. In distributed systems, it is crucial to establish an order of events that preserves their causal relationships, especially when events occur across multiple nodes without synchronized physical clocks. Logical time, distinct from real-time, provides such an order. It is represented using counters or sequence numbers, known as timestamps. These timestamps serve as abstract

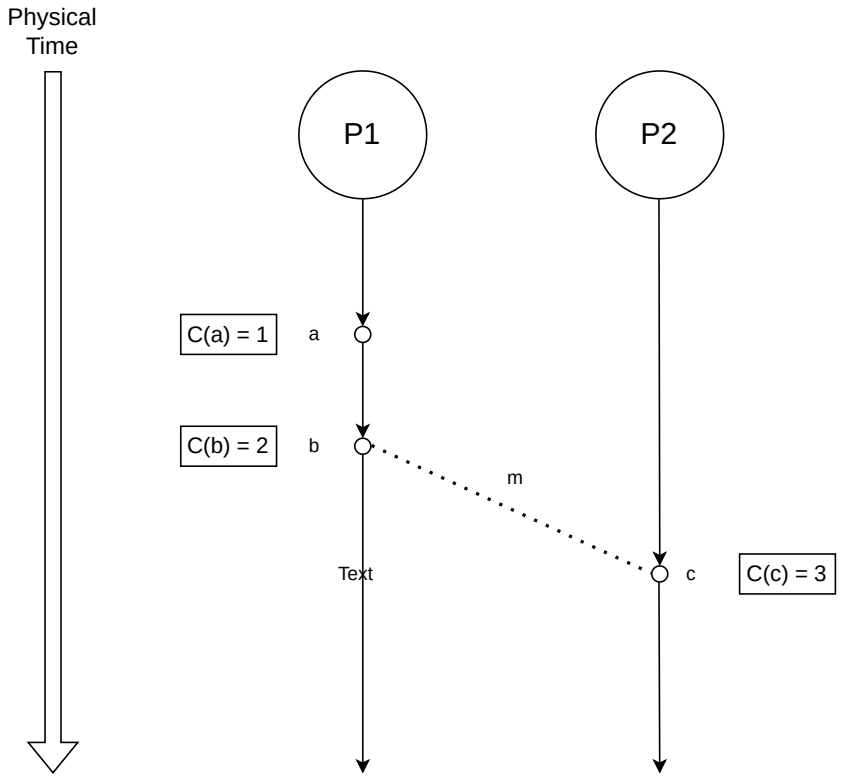

Figure 2.2. Lamport Clock Modell

representations of the "time" an event occurs in the system. An event's timestamp is determined by each process within a distributed system that maintains a counter. If an event occurs the process increments its counter in association with the event. Additionally, when a process sends a message it also increments its counter and sends that counter value alongside with the message. With the goal of ensuring total order, the recipient process, upon receiving a message, sets its counter to one greater than the maximum value between its current state and the timestamp of the received message.

There are 2 processes $P1, P2$, a clock C and three events a, b, c in Figure 2.2. At the beginning the clock for $P1$, written as $C(P1)$, and for $P2$ are zero. When event a occurs the clock is set to $C(P1) = 1$ but the clock for $P2$ remains unchanged. A message is send to $P2$ after b occurs and the clock is set to $C(P1) = 2$. At the event c of receiving the message the clock is set to $C(P2) = 3$, resulting in $C(P1) = 2$ and $C(P2) = 3$.

While logical time captures order and causality in distributed systems, hybrid systems, some systems demand a richer temporal representation: superdense time.

Superdense time is denoted as $\mathbb{R}^+ \times \mathbb{N}$ where $\mathbb{R}^+$ symbolizes continuous time, while $\mathbb{N}$ represents a discrete event sequence [Lam19]. We call $t = (a, b) \in \mathbb{R}^+ \times \mathbb{N}$ a tag, where a is the time value and b is the microstep [LRG+20]. Moreover, superdense time ensures a lexicographical ordering. Hence, if $(t_1, n_1), (t_2, n_2) \in \mathbb{R}^+ \times \mathbb{N}$ with $(t_1, n_1) < (t_2, n_2)$ then $t_1 < t_2$ or $t_1 = t_2 \wedge n_1 < n_2$.

Lets consider a more realistic example of a braking system. Modern vehicles, especially those with advanced driver-assistance systems (ADAS) features, are increasingly integrating both continuous and discrete monitoring and control mechanisms. If the driver pushes down on the brake pedal, a set of distinct actions take place. Specifically, the pedal sensor activates, and the brake lights illuminate, among other things. We adjust the prior example with a sensor in the front of the car, that triggers an automatic brake if the velocity and distance indicate a collision. In a system where the braking power is not binary but gradually, using only timestamps might lead to unintended behavior. If the event from automatic braking system and the event from the pedal occur at the same logical time, while they might be milliseconds apart from each other, there is no way of determining which event occured "before" the other one. This may result in over-braking if the inputs are naively combined or in under-braking if one input is ignored.

2.3 Reactors and Reactions

As mentioned earlier the actor model does not guarantee an ordering in which messages are received or processed. An LF program consists of one main top level reactor, which serves as an entry point for the execution. A reactor can be considered as a deterministic version of actors that consist of reactions, where reactions are not only message handlers but rather respond to discrete events. The reaction itself can also produce events on which other reactions in other or the same reactor are triggered. More precisely reactors only communicate over events, while events relate values to a tags [Loh20].

```
1 target Python
2
3 reactor BrakePedal {
4  output trigger
5
6  reaction(startup) {=
7   #read pedal data continously
8  =}
9 }
```

```
10
11 reactor Sensor {
12   input trigger
13   output data
14
15   reaction(trigger) -> data {=
16     # process data and send message
17
18     data.set(True)
19     //or false
20   =}
21 }
22
23 reactor Brakes {
24   input data
25   input trigger
26
27   reaction(trigger) {=
28     //react to trigger
29   =}
30
31   reaction(data) {=
32     //react to data
33   =}
34 }
35
36 main reactor {
37   bp = new BrakePedal()
38   s = new Sensor()
39   b = new Brakes()
40
41   bp.trigger -> s.trigger
42   bp.trigger -> b.trigger
43   s.data -> b.data
44 }
```

Listing 2.1. Simplified braking system

To address the example from Section Section 2.1, an LF program is created as seen in Listing Listing 2.1. This system does not meet the requirements and properties of the previously explained system. It is a conceptual framework for the system, and further functionality will be added in this chapter. Before delving into the code

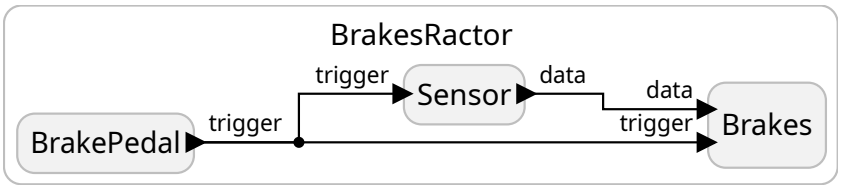

Figure 2.3. Brakes LF

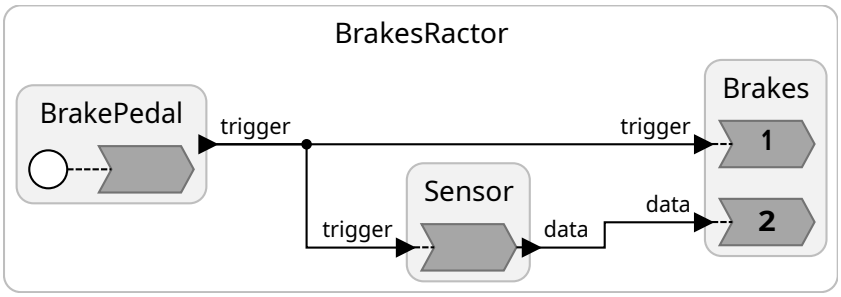

Figure 2.4. Brakes expanded LF

lets take a look at the diagram in figure Figure 2.3. This is the result of LFs runtime diagram generator. In order to leverage this feature either the VSCode plugin[3] or the Epoch IDE[4] needs to be installed on the machine. Figure Figure 2.4 is the same diagram but expanded through double clicking on the reactors. This allows for probing further into the internal structure of the system components which by itself can contain more expandable or collapsable sub components. A LF program starts with the declaration of the target language, as it is done here with python. I can contain several reactors, but must define a top level main reactor, as done at the bottom of listing Listing 2.1, with the intent of generating executable programs. Any additional reactor consists of a set of reactions that are routines initiated by an event. Events are provided as arguments for the reaction like it is done in listing Listing 2.1 with the keyword startup in the reactor `BrakePedal` and with the input trigger in the reactor sensor. Reactions can have multiple comma separated events, where only one of those events is needed to invoke the reaction.

Events that can invoke a reaction are inputs, keywords such as startup, and logical or physical actions. The **startup** keyword ensures that the reactions body is executed at the initialization of the reactor. LF reactors exclusively communicate over dedicated ports, inputs are ports for receiving messages. Outputs on the other hand can be used (as an event) to invoke reactions in a connected rector. If an input value is needed for some processing in the body of a reaction, but by itself should not trigger the reaction,

[3]www.lf-lang.org/docs/handbook/code-extension
[4]www.lf-lang.org/docs/handbook/epoch-ide

the input name is declared after the closing bracket for the reactions parameters. If a reaction is intended to modify the state of the reactor by affecting its outputs or modes, as will be explored in subsequent sections, it should be declared post the reaction parameters using an arrow, as illustrated in line 15. Should more than one output or mode be impacted, they can be specified through comma separation. To assign a value to an output port, this is accomplished as demonstrated in line 18. In this case a boolean is used but this is interchangeable for any type of the specified target language. Reactions share a common state within their reactor exclusively. The body of the reaction contains computations and operations that are intended to be executed as a response to a specific event. In the main reactor, which can has similarities to the main function in languages like C as an entry point for the program, reactor instances can be created in a object oriented fashion with the keyword new. The different ports of each reactor are connected as seen in line 41-43. Although the reaction body does not currently provide functionality, LF addresses the issue of actors not guaranteeing message ordering when they arrive simultaneously. First, LF orders messages or events by their tag. If an event occurs at the reactor Brakes on the data port at tag t_1 and an event at the port trigger at tag t_2, with $t_1 < t_2$ then the reaction triggered by the message on data is invoked first. The most distinguishable difference can be observed in the case of $t_1 = t_2$. LF schedules the invocation of the reaction based on the order in which they have been declared in the code. In this case, the reaction, triggered by event on the input port trigger is invoked before the reaction triggered by the data input port. This is also highlighted by the LF diagram in figure Figure 2.4, where reactions are displayed as flag shaped objects that also include the numbers one and two. This reflects the execution order discussed.

By modifying the program, adding LFs build in objects like tags as seen in the print statements of the listings Listing 2.2 - Listing 2.4, the time mechanism can be explored more in depths. In Listing 2.5 there is no advancement in the tag over the whole cycle of the running system, even though the physical time is advancing. In LF logical time is always chasing physical time [LRG+20]. The reactions behave as expected because the reactor Brakes receives both messages at the same tag, while maintaining the order of the reaction declaration.

With a minor tweak, the system now has a 100 milliseconds delay for the transmission of the message trigger from the reactor BrakePedal towards the reactor Brakes. This results in a different order, but as expected from LF, of invoked reactions. This and the difference of 100 milliseconds for both reactions in Brakes can be observed in Listing 2.6.

```
reactor BrakePedal {
   output trigger

   reaction(startup) -> trigger {=
      #read pedal data continously
      print(f"Startup:\nlogical
          time:{lf.tag().time}\
      \nmicrostep:{lf.tag().microstep}\
      \nphysical
          time:{lf.time.physical()}")
      trigger.set(True)
   =}
}
```

Listing 2.2. Reactor BrakePedal

```
reactor Sensor {
   input trigger
   output data

   reaction(trigger) -> data {=
      # process data and send message
      print(f"Sensor
          reaction:\nlogical
          time:{lf.tag().time}\
      \nmicrostep:{lf.tag().microstep}\
      \nphysical
          time:{lf.time.physical()}")
      data.set(True)
      #or false
   =}
}
```

Listing 2.3. Reactor Sensor

```
reactor Brakes {
   input data
   input trigger

   reaction(trigger) {=
      print(f"brakes
          trigger:\nlogical
          time:{lf.tag().time}\
      \nmicrostep:{lf.tag().microstep}\
      \nphysical
          time:{lf.time.physical()}")
   =}

   reaction(data) {=
      print(f"brakes data:\nlogical
          time:{lf.tag().time}\
      \nmicrostep:{lf.tag().microstep}\
      \nphysical
          time:{lf.time.physical()}")
   =}
}
```

Listing 2.4. Reactor Brakes

```
Startup:
logical time:1694339270372973022
microstep:0
physical time:1694339270373003644
Sensor reaction::
logical time:1694339270372973022
microstep:0
physical time:1694339270373025277
brakes trigger:
logical time:1694339270372973022
microstep:0
physical time:1694339270373033636
brakes data:
logical time:1694339270372973022
microstep:0
physical time:1694339270373041174
```

Listing 2.5. Terminal Output

```
Startup:
logical time:1694343258718668972
microstep:0
physical time:1694343258718708630
Sensor reaction:
logical time:1694343258718668972
microstep:0
physical time:1694343258718740321
brakes data:
logical time:1694343258718668972
microstep:0
physical time:1694343258718752659
brakes trigger:
logical time:1694343258818668972
microstep:0
physical time:1694343258818793562
```

Listing 2.6. Terminal Output after delay

2.4 Timers, States, Actions, Preambles and Composition

The running example is extended to serve more functionality while showcasing different features of LF. Please keep in mind that the design choices are primarily made to motivate and explain different features of LF. This is by no means a system architecture of a safe brakeing system rather than a collection of design ideas a developer could come across while building a similar system. Thus, the reader should be able to transfer this ideas to actual one of its own system.

In Figure 2.5 is the LF diagram for the extended program. The program simplifies an actual brakeing process by rather applying a "all or nothing" or in this case "brake or don't" approach. In reality, the intensity of the application of the brakes has much higher status that in this design. The old reactor Signal is now renamed to Signals since it contains itself two other reactors, SensorFront and Sensor back. The reactors BrakePedal and Brakes now have more reaction. The system now fulfills more requirements and adds a front sensor for automatic brakeing, which is seen more often in actual applications.

The Sensors reactor itself has three reactions of its own and two composed reactors. Sensors front has a timer which triggers the reaction, that probes the current distance to the next obstacle in front of the car and sends the value to the reaction of its parent reactor Sensors. The reaction then, computes the stopping distance with the given

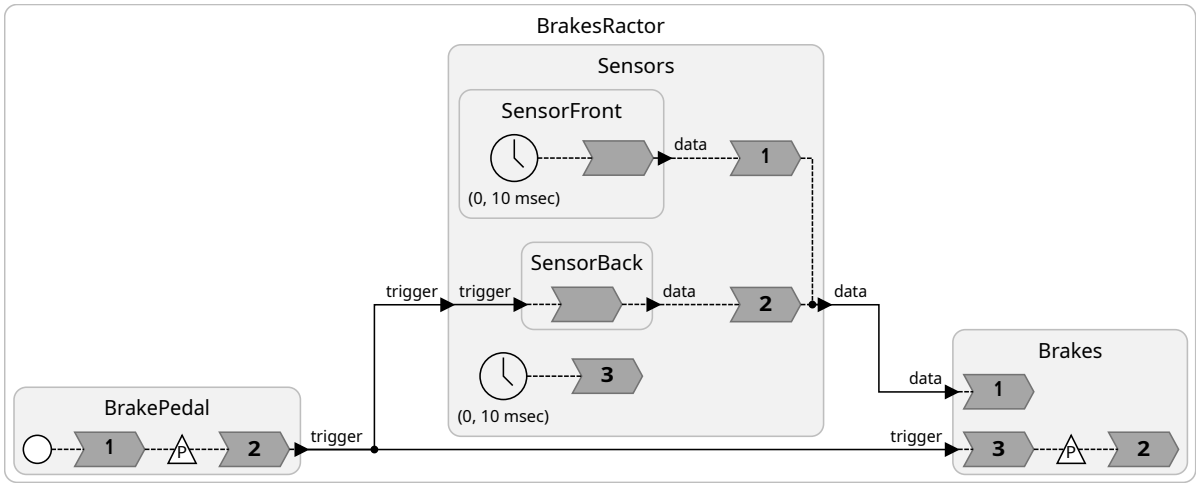

Figure 2.5. More detailed version of the running example

velocity and decides if the current distance towards the next obstacle requires a full brake and in this case sends a message to the data port of Sensors and therefore to the data port of Brakes. The SensorBack's port trigger is connected to the trigger port of Sensors. This means, that if a braking signal arrives from the driver, or in this model from the reactor BrakePedal, at Signals, it is in fact handled by the SensorBack reactor first. There it is assessed whether a full stop is justified, taking into account the current speed and the calculated stopping distance. The result is sent to the data port of the Sensors reactor, hence also to the input port data of the reactor Brakes. The Brakes reactor handles the incoming messages. If a messages is received on port data the message could be a brake signal from SensorFront or from SensorBack. If the SensorFront reactor sends a brake signal, a full stop is applied because otherwise a the car is at risk of a collision with the obstacle in front of the car. This is in case of same time stamps always the first priority. When a message from SensorBack is received, it indicates that there is a vehicle too close to the back of the car that is at risk of a collision at full brake. Therefore, a less intensive brake is applied. The manual brake by the driver might be premature to avoid a collision with the obstacle in front. It is important to note, that if the reduced brakeing would result in a risk colliding with the obstacle in front of the car, the full brake signal would always be of higher priority, since it is assumed that a potential rear-end collision is of lower priority than colliding with an obstacle in front. A manual brake from the driver triggers reaction three in Brakes. This sets up a physical action which acts in this example as a timer of ten milliseconds. After that ten milliseconds mark, reaction two is triggered resulting in a full brake since there is no evaluation from the SensorBack reactor available at this moment. It is considered a higher priority that the car brakes are applied fully

than not being applied at all. The code in Listing 2.7 - Listing 2.10 shows how the system is build. All instances of random number generation would be exchanged for real sensor data in a cyber physical system or with test data in system and unit tests. The same is true for the print statements in the Brakes reactor.

```
reactor BrakePedal {
  preamble {=
    from threading import Thread
    def read_sensor(self,
        brakeing_signal):
      while(True):
        pedal = input("Press enter for
            brakeing")

        if pedal == "":
          brakeing_signal.schedule(0)
  =}
  output trigger
  physical action brakeing_signal

  reaction(startup) ->
      brakeing_signal {=
    t = self.Thread(
    target=self.read_sensor,
    args=(brakeing_signal,))
    t.start()
  =}

  reaction(brakeing_signal) ->
      trigger {=
    trigger.set(True)
  =}
}
```

Listing 2.7. Terminal Output

```
reactor SensorBack {
  preamble {=
    import random
    def get_distance(self,):
      #meters
      return self.random.randint(0,100)
  =}
  input trigger
  output data

  reaction(trigger) -> data {=
    data.set(self.get_distance())
  =}
}

reactor SensorFront {
  preamble {=
    import random
    def get_distance(self,):
      #meters
      return self.random.randint(0,100)
  =}

  output data
  timer t(0, 10 ms)

  reaction(t) -> data {=
    data.set(self.get_distance())
  =}
}
```

Listing 2.8. Terminal Output

The reactor BrakePedal introduces at the beginning the **preamble** keyword. This allows to define code or even load external libraries into a LF program. A function called read_sensor is defined, so that it can be used in a thread later on inside the logic of LF. The benefit is, that code or even the code of extensive libraries, don't need

to be declared inside of reactions and reactors. Code that implements the the logic of reading sensor data can be written in an external file and used inside of reactors and reactions, hiding and abstracting away information. Thus, the **pramble** keyword offers modular development and more readable code and lets the developers focus on the orchestration of the system while using already developed functionality.

Timers, as defined in line 10 of Listing 2.8, are used to to manage and schedule recurring tasks, a scenario frequently encountered in embedded computing systems. These timers are initiated with a specified delay, referred to as the offset. If the offset is set to zero, the timer triggers at the logical beginning of execution, following which, it activates at regular intervals defined by a specified period [LMB+21]. In LF, timers trigger events, which trigger reactions as seen in line 34 of Listing 2.8. Here the timer is scheduled so that the velocity is read every ten milliseconds. In safety critical systems, especially in those with low computing power, busy waiting implementations may not be the best choice. Other known concepts such as hardware interrupts or function callback offer a more resource friendly alternative.

Sometimes an event needs to be scheduled in a non periodic manner, different to how it is realized with timers. In the given example after the reactor Brakes received a signal on trigger, it should wait for a specified period of time for a signal on the data port in order to evaluate if it is safe to brake. If for some reason, reaction one in Brakes is not triggered in the specified amount of time, the brakes are applied fully. But this time specific event is usually not occurring regularly in the same time interval. For tasks like this LF introduces **logical actions** and **physical actions**. Logical actions are scheduled synchronously inside the body of a reaction at the tag t. The scheduled event is triggered at $t' = t + d_{min} + d$ where d_{min} is the minimum delay and d the additional delay argument [LMB+21]. Additional arguments can be used at declaration to invoke specific constrains for the specified logical action as seen in the LF handbook[5]. In general d_{delay} is zero. The scheduling of a logical action is done with the invocation of the build in schedule method, which expects an argument, specifying the delay that results in t' as the tag at which the event occurs. If $d = 0$, the event does not occur at tag $t = (x, n)$ but one micro step later at $t' = (x, n + 1)$ in order to obtain determinism [LMB+21]. It is important to note that this scheduling may not provide the responsiveness with regards to physical time, as it is often expected in real time systems.

If the timing of an event is not determined by the reactor but by the physical environment, physical actions provide an asynchronous scheduling mechanism. This is the case in the running example. The sensor data resulting from the application of the brake pedal is an external event. Physical actions are declared using the keyword

[5]https://www.lf-lang.org/docs/handbook/actions

physical instead of **logical** as seen in line 12 of Listing 2.7. The action braking_signal is scheduled not in the reaction itself but in an additional thread running the read_sensor function, in order to react to a brake signal modeled as user input from the keyboard. If there is an input from the user present the physical action schedules asynchronously an event which then triggers reaction two in the reactor BrakePedal. At the scheduling of a physical action there is no current logical time. Therefore, the scheduling of the action occurs at the tag $t' = T + d_{min} + d$, where T is a measurement of the relative physical time of the host system. By allowing external events to influence the behavior of the LF program, one may conclude that this affects the determinism of the system. That is not the case if one considers the assigned tags to those events as an input of the program itself [LMB+21].

State variables are shared among reactions but not between reactors. They provide a shared state for different reactions on which the body of a reaction may depend [Loh20]. In the running example the state variable **driving** is declared in Listing 2.9 line 11. Here it is assigned to the integer 0. In general, state variables can be declared as any type provided by the target language. In this case it serves to check whether the car is actually driving before initiating sensor reading or automated brakeing signals. It can be used the same way as it is done often in development as a conditional in order to invoke functionality based on the current state the software system. The use of state variables in that sense is further discussed in Chapter 3.

In LF reactors can be created inside of other reactors, as it is seen in the Sensors reactor, adding modularity and encapsulation.

```
reactor Sensors {
  preamble {=
    import random
    def get_velocity(self,):
      # meters per second
      return self.random.randint(0,35)
  =}
  input trigger
  output data
  timer t(0, 10 ms)
  state driving = 0

  front = new SensorFront()
  back = new SensorBack()

  trigger -> back.trigger

  reaction(front.data) -> data {=
    if self.driving > 0:
      stp_dist = (self.driving *
          self.driving) / 20

      if (stp_dist - front.data.value
          < 0):
        data.set(True) #brake
  =}

  reaction(back.data) -> data {=
    if self.driving > 0:
      stp_dist = ((self.driving *
          self.driving) / 20)/2

      if (stp_dist - back.data.value <
          0):
        data.set(False)
  =}

  reaction(t) {=
    self.driving = self.get_velocity()
  =}
}
```

Listing 2.9. Terminal Output

```
reactor Brakes {
  input data
  input trigger
  physical action wait

  reaction(data) {=
    if data.value:
      print("Full brake due to front
          sensor")
    else:
      print("brake with half agents
          intensity due to back sensor
          data.")
  =}

  reaction(wait) {=
    print("brake with agent intensity.
        No data from back sensor.")
  =}

  reaction(trigger) -> wait {=
    self.brakeing= True
    wait.schedule(MSEC(10))
  =}
}
```

Listing 2.10. Terminal Output

2.5 Modal Reactors & Deadlines

The requirements of complex software systems are often to operate under distinct scenarios in different ways. In the running example, one could argue that certain code is only executed if the car is driving and so on. Naively, this can be approached by invoking specific code under the condition of the system being in a distinct state, as proposed in the prior section. For large systems this can become a complex and inconvenient task. In LF, modal reactors are introduced to reduce implementation complexity while providing more readable code.

Modal reactors in LF extend the existing reactor model by modal coordination. This is achieved by segmenting reactors and reactions into disjoint subsets of mutually exclusive modes. A reactor containing modes is called a modal reactor, where only one mode can be active at a specific logical time instant rendering other modes inactive. The concept of modal models is not new but towards a polyglot modal coordination layer guided by LF, it is [SHL+23].

In the running example the velocity is measured every 10 milliseconds. Even if the messaging of the Sensors reactor depends on the value of the state variable, the front sensor is active as long as the system is running. Considering a scenario where the car is in a parking lot or waiting at a traffic light, it is unnecessary for the front sensor to be active and for the speed measurement to be taken in such a short interval. The extended system should only read sensor data if the car is actually moving and relax the period of time for measuring the velocity if the car is not moving.

Modes can be defined within any reactor for which a unique name is required and may contain elements local to them like state variables, timers, actions, reactions and even reactors. Hierarchical composition is possible through the instantiation of modal reactors although modes can not be nested directly within other modes. Mode transitions are declared within reactions and can trigger a switch to another mode. Transitions are categorized into reset or history transitions where reset transitions reinitialize the mode while history transitions retain the me state of the mode from its last activation. If a reaction triggers a mode switch, it needs to be declared as an effect as seen in Listing 2.11 line 15 and 42, with the keyword **reset** or **history**. Mode execution is confined to its active period, and upon mode transition, execution in the current mode ceases and begins in the new mode one microstep later. The concept of local time is employed in modes to govern the execution of time-related components, with the time progression suspended in inactive modes. In a modal reactor there needs to be an initial mode declared, that is active at the startup of the reactor. Reactions within modes can apply the trigger **reset** that ensures that the reaction is invoked every time the mode becomes an active one [SHL+23].

In Listing 2.11 and Figure 2.6 the initial mode is stop at startup. The velocity is

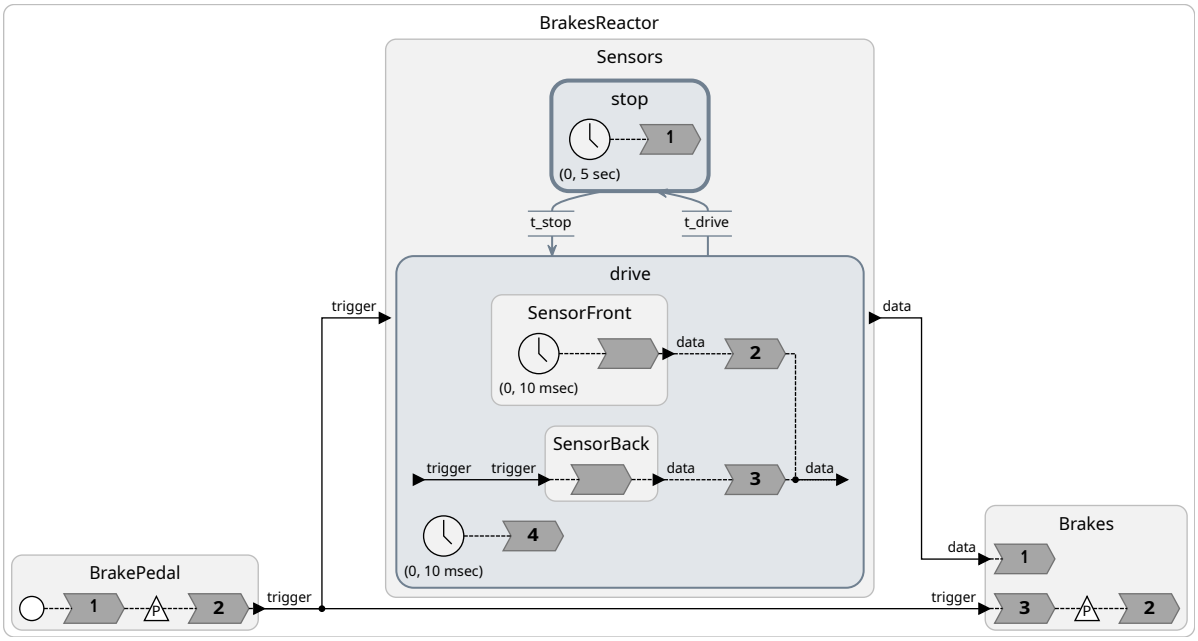

Figure 2.6. Sensors reactor as modal reactor

sampled every five seconds and neither sensor data nor the trigger input port has an effect while being in that mode. As soon as the car moves and a velocity greater than zero is detected, the reactor transitions into the drive mode. There, the already known functionality of sensor reading and messaging is applied with the one change that if the car stops (velocity is zero) the reactor transitions back to the stop mode.

Furthermore LF includes the concept of deadlines. The reader might think that bounds like deadlines can be used in or running example as a guarding mechanism that triggers functionality if a deadline is violated. For example, if the car detects a potential collision with an obstacle, it might warn the driver first instead of autonomously applying the brakes. If the driver does not react within the constrains of the deadline, the brakes are applied. A deadlines D in LF is a bound on the maximum latency between sensing and actuation, meaning that the reaction to an input with tag t is required to be invoked before the physical time $t + D$ on the host system is exceeded. This makes the concept of deadlines lazy, since the deadline violation is only detected if the reaction is invoked at its appropriate logical time. Before a reaction with a deadline is invoked the LF runtime checks if the local physical time T is smaller than $t + D$. If that is true, the reaction is handled as usual but if not the body of the deadline will be invoked [Loh20].

This makes deadlines in LF not suitable for the proposed example since the detection of the violation might appear to late. Therefore, deadlines in LF serve two

purposes: As as a fault handler to implement behavior, such as returning into a safe mode of operation, if a deadline is violated, and secondly as information for the scheduler of LF to prioritize reactions and its downstream reactions with a smaller deadline interval [LMB+21]. It is important to note, that the application of deadlines in LF admits nondeterminism, since the execution of the reaction or deadline body depends on factors outside of the system semantics of LF[Loh20].

This section does not provide an explicit example of the syntax of deadlines since the concept is used in Chapter 4 and further complication of the running example seems unnecessary.

```
1  reactor Sensors {
2   preamble {=
3     import random
4     def get_velocity(self,):
5       # meters per second
6       return self.random.randint(0,35)
7   =}
8
9   input trigger
10  output data
11  state driving = 0
12
13  initial mode stop {
14    timer t_stop(0, 5 sec)
15    reaction(t_stop) -> reset(drive) {=
16      self.driving = self.get_velocity()
17      if self.driving > 0:
18        stop.set()
19    =}
20  }
21
22  mode drive {
23    front = new SensorFront()
24    back = new SensorBack()
25
26    trigger -> back.trigger
27    timer t_drive(0, 10 ms)
28    reaction(front.data) -> data {=
29      stp_dist = (self.driving * self.driving) / 20
30
31      if (stp_dist - front.data.value < 0):
```

```
32        data.set(True) #brake
33      =}
34
35      reaction(back.data) -> data {=
36        stp_dist = ((self.driving * self.driving) / 20)/2
37
38        if (stp_dist - back.data.value < 0):
39          data.set(False)
40      =}
41
42      reaction(t_drive) -> reset(stop) {=
43        self.driving = self.get_velocity()
44        if self.driving == 0:
45          stop.set()
46      =}
47    }
48  }
```

Listing 2.11. Modal reactor Sensors

2.6 Distributed Execution

So far all examples may occur as distributed components due to the different reactors involved, while in fact they run, if executed, on a single CPU. But in this thesis LF was not only advertised as a tool for building concurrent systems, but also for building distributed systems that assure determinism. The running example is still used in this section but without providing specific source code or diagrams since the syntactical changes required to achieve distribution are minimal. Chapter 4 provides source code for distributed reactors and an example of a more realistic distributed system in LF can be found in [LMS+20].

In order to discuss how distributed execution works in LF the running example is simplified from the last implementation in Section 2.5. The system still consists of three reactors. The first reactor integrates a physical input from the press of the brake pedal. The second reactor uses sensor data from the rear of the car to determine if it is safe to brake and the third reactor enforces the brakes to apply if appropriate. If the pedal is pressed a message is send to the sensor reactor and to the Brakes reactor. Although, some changes are made. As the sensor reactor receives a message it checks if it is safe to break but only sends a message to the brake reactor if it is not. If the Brakes reactor receives a message at the lower communication path from the brake

pedal reactor it should wait for a specific amount of time for a message from the sensor reactor before it engages in a full brake. If it receives a message from the sensor reactor it applies a smoother brake in order to avoid a rear collision. If it does not receive a message from the sensor reactor within a specified time frame a full brake is applied.

In the model running on a single machine, this was achieved by using physical actions and ordering of reactions in the Brakes reactor. In the distributed version the Brakes reactor has two reactions. The first reacts to the input of the sensors and the second to the input of the brake pedal reactor. This ensures that if both messages are received simultaneously with respect to the logical time of the system, that the smoother brake is applied before the other one. But in a car those components would usually not run on the same machine and rather be different distributed components. In LF, if a top level reactor is declared with the keyword **federated** instead of **main**, the runtime generates a separate executable for each reactor instantiated inside of the top level reactor. The coordination of the federate can be centralized or decentralized, which needs to be declared at the beginning of the top level LF file right after the declaration of the target language [LMS+20].

New challenges arise if the running example is executed as a federate. If the pedal is pressed at the physical time T a physical action triggers the downstream reactions. The reactions are invoked at the same tag t for which $T = t$. Thus, the reactions at the Brakes reactor see their input logically simultaneous at tag t. The ordering of the reactions would guarantee that the reaction triggered by the message from the reactor sensors is invoked before the reaction triggered by the message from the reactor Brakes. Even if the messages are received logically at the same time, this does not mean that they are received at the same physical time in a distributed system. Aspects such as Latency, execution time or clock synchronization errors may interfere with the system. For example, the brake reactor might receive the message from the brake pedal reactor before the message of the sensor reactor over a network due to latency issues making it error prone and resulting in a fully applied brake even if it was not safe to brake. The message from the sensors reactor is received later with regards to physical time while having the still the same tag as the message from the brake pedal reactor.

Conserving logical time for maintaining an order in which those messages are received is not enough because in the physical world there is no "before" or "after" for two or more geographically distinct events. There is only an order of those events relative to the observer [LMS+20]. The problem exists further even if the brake reactor is chosen as the observer, because we have seen it could receive the messages out of order.

To address this and assure determinism it is important that the tags are preserved

across the network. Messages need to transmit their tags along with the messages and it must be avoided that a federate advances its logical time before it has seen all messages with the current tag. For this LF implements as already mentioned two different kinds of distributed execution, centralized and decentralized. In centralized execution, LF uses a another component, the Run Time Infrastructure (RTI). Each federate communicates with the RTI to share the earliest logical time at which it may send a message and to consult if it is safe advance its logical time. However, while this approach offers a straight way to address the problem of messages being received in the wrong order, it becomes a bottleneck with regards to performance, since every message needs to pass through the RTI. Additionally, it creates a single point of failure for the system and might slow down the systems advance of logical time making it less responsive [LMS+20].

Decentralized execution uses a technique called Programming Temporally Integrated Distributed Embedded Systems (PTIDES) [LMS+20]. This technique extends the discrete-event model for distributed real-time embedded systems, ensuring deterministic behavior through a well-defined order of event processing. It utilizes timestamps to establish a logical order of event processing, enabling determinism and repeatability in system behavior, while employing dependency analysis to ascertain safe processing sequences, allowing for effective handling of event dependencies in a distributed setting. Moreover, it incorporates synchronized physical clocks to align the logical timelines across distributed components, thereby facilitating correct order of event processing and addressing network latency issues. It also provides a framework that navigates the challenges of network latencies, clock synchronization errors, and distributed coordination, rendering a predictable and analyzable system behavior essential for real-time embedded system applications [ZLL07].

How this works can be explained referring to the running example. Let's say, the brake pedal is applied at the physical time $T = t$ where t is the assigned tag. We now quantify execution time (X_i), latency (L) and the clock synchronization error (E) by making assumptions about their bounds. These vary from system to system. Lets say, that the upper communication path, or rather the execution time of the reaction in the sensors reactor is X_1 and the one of the lower path is X_0. Hence, the messages of the sensors and brake pedals reactors are sent through the network at the latest of the physical time $T + X_1$ and $T + X_0$. If the latency of the network is L than the `Brakes` reactor receives the message from the brake pedal reactor no later than $T + X_0 + L$ and the message from the the sensor reactor no later than $T + X_1 + 2L$ (because of the two network connections). Assuming E is the bound for the clock synchronization error is E, then the messages arrive no later than $T + X_0 + L + E$ and $T + X_1 + 2L + E$ at the brake reactor. With that it can be determined that an input with tag t is safe to process if the physical clock of the brake reactor reaches

$STP = t + max\{T + X_0 + L + E, T + X_1 + 2L + E\}$. Now the reactor can engage in a full brake if no message was received from the sensors reactor within that time and a fault handler can be applied if a only a message was received from the sensors reactor withing the given time range.

Chapter 3

Concepts & Implementation

In this chapter the the systems of S&B are discussed even further. The diagram at Figure 3.1 shows a simplified version of how some of their products work together. For this chapter the focus lies on the iBP but even in more detail on the iUZ and iBS-Z. The the reason for the multiplicity of some components as well as components that have not been discussed so far, displayed in that diagram are not explained in this chapter but briefly elaborated in Chapter 4. Those distributed systems offer a variety of services as mentioned in the introduction. A user can make a time range based request from the iBP and if the request is valid the iBS-Z returns recorded log events for the system from a database for that time range.The request is processed by two sub components: HistoryStateMachine which is part of the iBS-Z and SimuHistoryStateMachine which is part of the iUZ. Both sub components are modeled with SCChart as displayed in Figure 3.2 and Figure 3.3. They work concurrently but communicate with each other. The user interface of the iBP is not part of the consideration in this thesis, but is rudimentary implemented in LF for the purpose of creating a working system. Some path labels at the SCChart diagrams contain elements like ia16520. Those are telegrams that are sent between HistoryStateMachine and SimuHistoryStateMachine and contain state specific information. Lets discuss how a user request is processed in both components. The description follows the process along the transitions and states as depicted in the SCChart diagrams. Some functionality is note elaborated if it is not important for the rest of this thesis or is obvious from the diagrams. At default, both components are in the Idle state. If a user makes a request from the iBP, it is send to HistoryStateMachine. The request message is then forwarded to the SimuHistoryStateMachine via a ia16520 telegram and the system transitions into the top level state ValidateRequest and right after into the Start state. There, HistoryStateMachine waits for a response from SimuHistoryStateMachine. If the telegram reaches the SimuHistoryStateMachine it transitions from into the top level state Processing Request and afterwards into the composed Start state. From there, the component checks if the requested date time range is valid. An invalid range would be one where the start point is larger than the end point. If invalid, the SimuHistoryStateMachine sends a ia16020 telegram to the HistoryStateMachine containing information about of the abort transitions into the Abort state, followed by the Done state and eventually ends in the Idle state again.

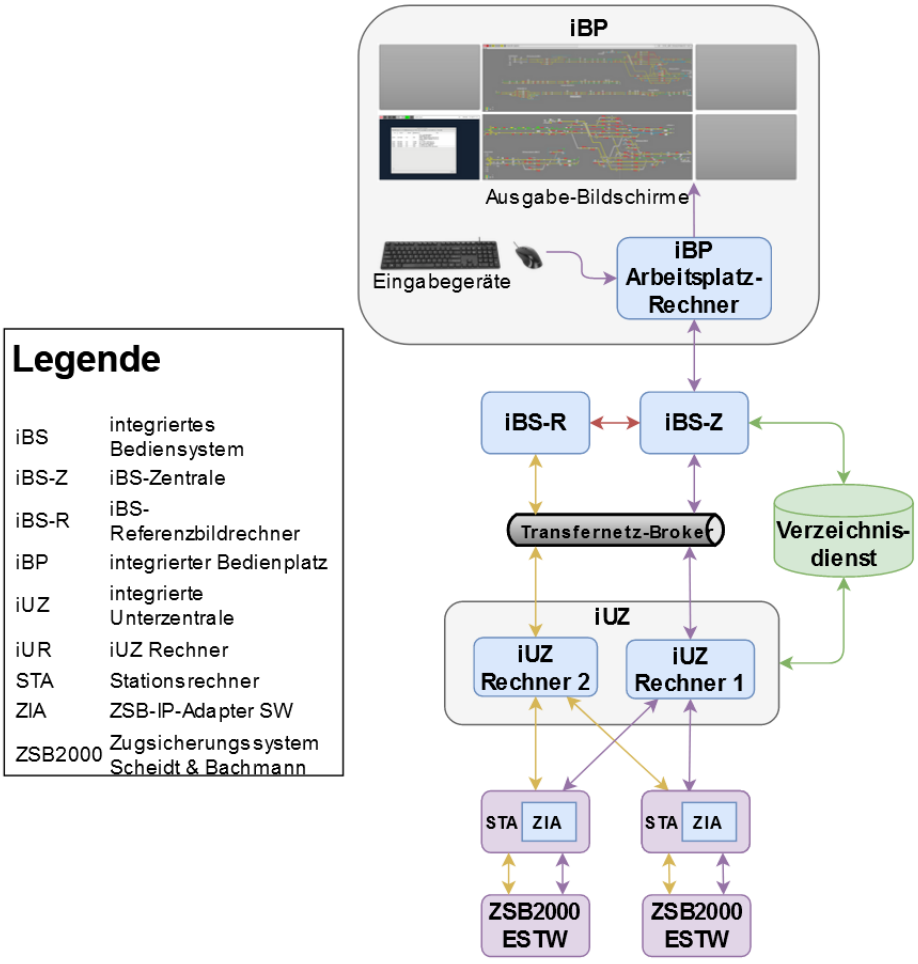

Figure 3.1. Overview of different systems at S&B

If that happens and the HistoryStateMachine receives the ia16020 telegram if follows a similar transitional path into AbortRequest, then the Done state and eventually the Idle state. In case of a valid time range SimuHistoryStateMachine, transitions into the CheckMessageCount state, where the amount of log data of the given time range is sampled and checked for a valid size. The size can be zero entries resulting for SimuHistoryStateMachine and HistoryStateMachine in a similar abort transition path as before. If the amount is higher than a specified cap value, SimuHistoryStateMachine sends a ia16020 telegram to HistoryStateMachine, that contains the information of the requested amount of log data being larger than the allowed maximum. Additionally, SimuHistoryStateMachine stats an internal timer. When HistoryStateMachine gets notified of that, it sends a message, resulting in a popup for the user at the iBP. The user is

asked if he actually wants to see that amount of log data. Both components follow a similar abort path as before, sending and receiving a ia16020 telegram and eventually transitioning it the Idle, if the user does not respond within the bounds of the timer of SimuHistoryStateMachine. If the user response is negative, HistoryStateMachine gets notified of that, sends a ia16530 telegram to SimuHistoryStateMachine and transitions eventually into the Idle state. When SimuHistoryStateMachine aborts the process due to the content of the telegram, it stops the timer and transitions eventually back into the Idle state. In the case of a positive response from the user, HistoryStateMachine still sends a ia16530 telegram to SimuHistoryStateMachine but with different content and transitions into the AwaitiuzReply waiting for a response from SimuHistoryStateMachine. SimuHistoryStateMachine responds to the telegram with yet another ia16020 telegram informing HistoryStateMachine, which transitions into the DisplayRequest state. SimuHistoryStateMachine itself transitions into the SendDokuMessages state. From there, it sends the log date in chunks via a ia16030 telegram if necessary and transitions afterwards through the Done state back into the Idle state. The HistoryStateMachine component checks the response for consistency and displays it for the user in the iBP if no error occurs, and transitions eventually into the Idle state. Even in case of an consistency error, the final state will be Idle, but without displaying any log data to the user. One path of each component have not been discussed yet. If SimuHistoryStateMachine is in the CheckMessageCount state with a allowed amount of log data being requested, both components behave as before if the requested data is send from SimuHistoryStateMachine and displayed at SimuHistoryStateMachine. Each path leads to the idle state for both components eventually making them accessible for another request from a user. Both diagrams also depict that if they are in their related date processing state, ValidateRequest for HistoryStateMachine and ProcessingReques for SimuHistoryStateMachine, can transition back into the Idle state with now regard of the current composed state they are. This can happen if the client aborts the request during the processing phase or if the system of the client restarts.

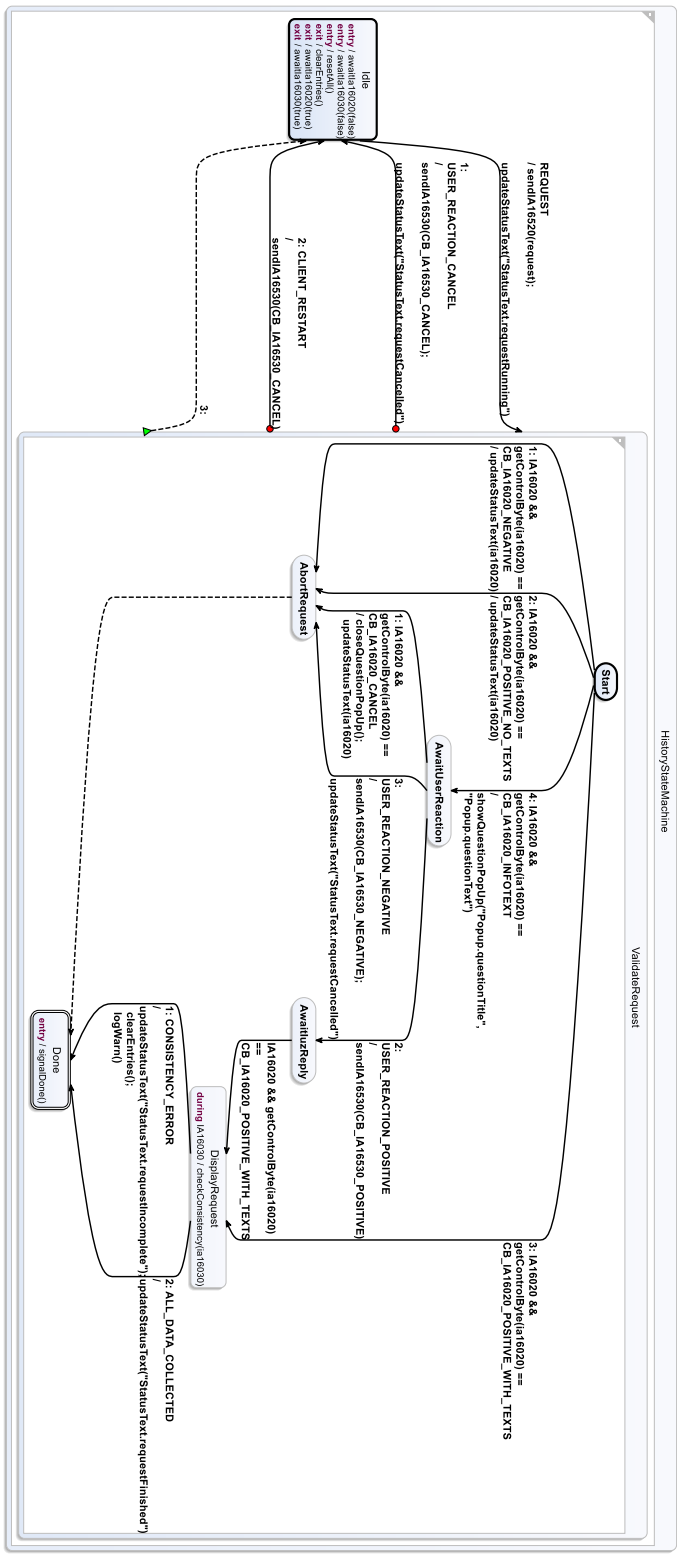

Figure 3.2. History state machine with SCChart

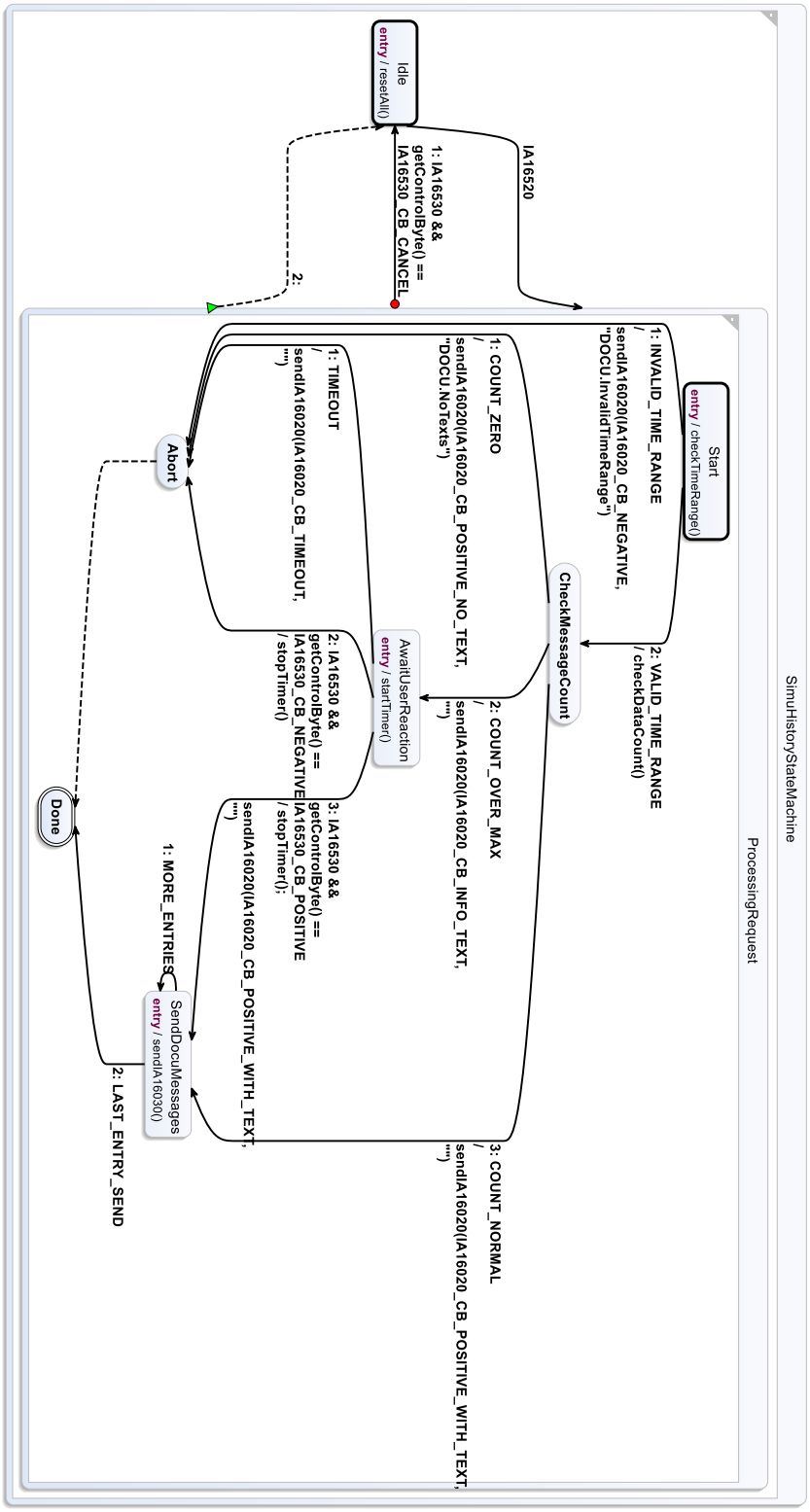

Figure 3.3. Simu state machine SCChart

3.1 Conceptional considerations

The goal is to rebuild both components in LF on a mostly conceptual but functional level. This means that some components or behaviors are not exactly replicated. For example, the "empty" states, such as abort or Done in SimuHistoryStateMachine are ignored since there is no functionality, important for the work of this thesis in those states. They exist due to the requirement specifications of the clients of S&B. Furthermore the states Start and CheckMessages in SimuHistoryStateMachine, are merged together since there is no need in the LF implementation to separate the validity checks into different states. Additionally, a data base is mocked and therefore log responses are not sent in chunks which renders consistency checks unnecessary.

The aim is not to simply rebuild the components in LF but rather building them explicitly from the SCChart notation, maintaining the sequential state driven behavior. For that three different approaches are used. All approaches may not be sensible in an usual exercise of building systems in LF. The idea is to subscribe as strictly as possible to a specific pattern of mapping the state driven behavior to a LF program. Please keep in mind that some design choices are the result of that strict pattern choices or the lack of experience of the author.

At the first implementation all state like behavior is realized using only state variables. This might appear naturally to some readers if the behavior of a system in specific states is controlled without the use of more sophisticated tools, than the chosen programming language has to offer on a basic level. To achieve a more modular sense of the states, the second approach uses a mixture of composed reactors and state variables. The composed reactors represent the states in each of the components. This makes it necessary to add new components that are not required in the actual system. Lastly, the third approach uses modes and composed modal reactors only without state variables. Figure 3.4 shows the system as a whole. The implementation of the top level main reactor does not change throughout all three design choices. The only difference is in the internal implementation and behavior of the Simu and History component. How the Gui component is implemented is not important. It only serves as simplified part of the user interface in the iBP. Both reactors have input and output ports, that represent the exact same telegram types as in the original system (e.g.ia16020). The request port and cancel port simulate the behavior of an user making a request or canceling as seen in Figure 3.2 on the transitional paths between Idle and ValidateRequest. Additionally the popup and data ports represent the HistoryStateMachine opening a popup for the user in case the request was to large and sending data to the user for display in case of a valid request. In the original system if the timeout bound is violated, SimuHistoryStateMachine notifies HistoryStateMachine which results in both components returning into the Idle state. HistoryStateMachine

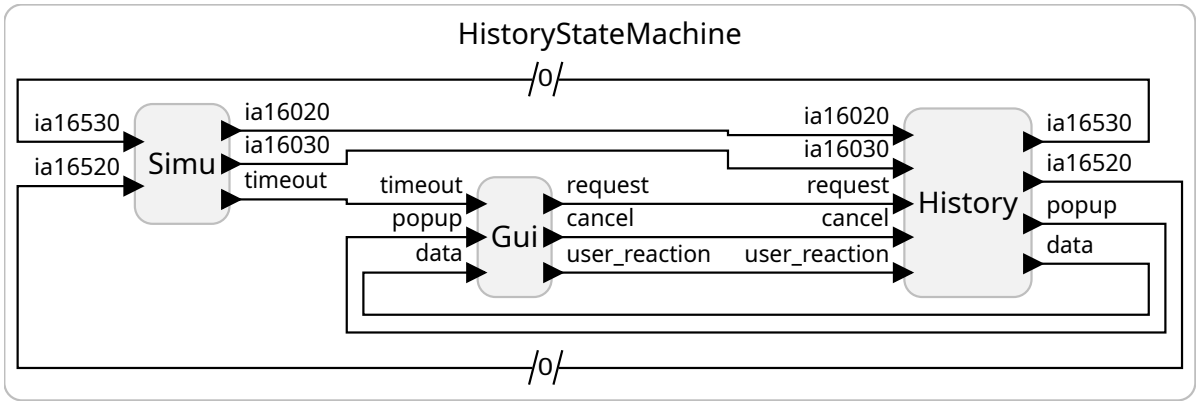

Figure 3.4. Top level reactor of HistoryStateMachine.lf

cancels the popup request at the iBP. This is slightly different in this implementation, because SimuHistoryStateMachine sends a message on the timeout port directly to the Gui. Everything else stays the same. At last, the user_reaction port is only used to transmit the message of a positive or negative reaction from the user at the popup. Some ports may seem , but they exist with the purpose do distinguish the different messages between the components more clearly especially with regards to the evaluation of the diagrams by developers. At two connections, there is a zero-shaped interruption. Those are realized with the keyword **after** This is called a logical time delay that causes a produced output to be delayed for a specified (logical) time before appearing as an input at the connected reactor [Loh20]. In this case, the time delay is zero (in the system one microstep later), ensuring that there are no cyclic dependencies between reactions.

```
target Python {
  keepalive: true,
  files: ["../external/telegrams.py", "../external/db.py", "../external/gui.py"]
}

import Simu from "Simu.lf"
import History from "History.lf"
import Gui from "Gui.lf"

main reactor HistoryStateMachine {
  simu = new Simu()
  hist = new History()
  gui = new Gui()

  hist.ia16520 -> simu.ia16520 after 0
```

```
16  hist.ia16530 -> simu.ia16530 after 0
17
18  simu.ia16020 -> hist.ia16020
19  simu.ia16030 -> hist.ia16030
20
21  gui.user_reaction -> hist.user_reaction
22  gui.request -> hist.request
23  gui.cancel -> hist.cancel
24
25  hist.data -> gui.data
26  hist.popup -> gui.popup
27  simu.timeout -> gui.timeout
28 }
```

Listing 3.1. Source code top level reactor

Since the three different implementations are discussed, there is no step-by-step description of the code unless it is important or not self-explanatory from the code or diagram itself. The diagrams serve primarily as a reference to the behavior of the implementations. The second implementation is discussed on a conceptual level because the amount of source code would not add sufficient understanding. Furthermore, all different implementations contain imported functionality from telegrams.py and db.py. Those provide a simplified version of the given telegrams and their contained information, and basic operations for the mocked data base which will not be further discussed for obvious reasons.

3.2 Implementation: State Variables

In Figure 3.5, the History reactor has a dedicated reaction for every input port and the possible states the reactor can be in: Idle, Start, AwaitUserReaction, AwaitIuzReply and DisplayRequest. There is only one state in which HistoryStateMachine can receive a request message, resulting in a reaction that only checks (even if absolutely necessary in this case) if the current state is correct, sends the appropriate telegram and changes the state variable to Start, as seen in Listing 3.2. The ia16020 telegram can be processed by HistoryStateMachine in multiple states. This is displayed in Listing 3.3. For example, if the reactor is in state Start and receives a negative response from Simu, the reaction changes the state to Idle. On the other hand, if the reactor is in state AwaitUserReaction and receives a positive response, it changes its state to AwaitIuzReply. This is similar for the rest of the reactor.

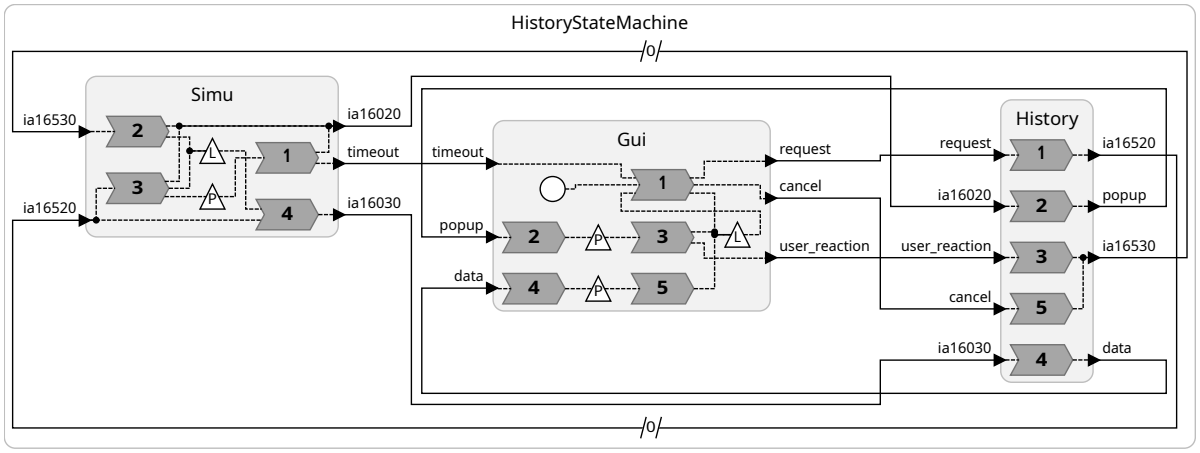

Figure 3.5. Expanded top level reactor of state variable approach

```
reactor History {
  preamble {=
    import telegrams
    tele = telegrams.Telegrams()
    states = ["Idle","Start",
    "AwaitUserReaction","AwaitIuzReply",
    "DisplayRequest"]
  =}

  input cancel
  input request
  input ia16020
  input ia16030
  input user_reaction
  output ia16520
  output ia16530
  output data
  output popup
  state st = "Idle"

  reaction(request) -> ia16520 {=
    if self.st == self.states[0]:
      ia16520.set(request.value)
      self.st = self.states[1]
  =}
```

Listing 3.2. Source code states History 1/4

```
reaction(ia16020) -> popup {=
  if self.st == self.states[1]:
    if ia16020.value ==
        self.tele.ia16020[1]:
        #ia16020_CB_NEGATIVE
      self.st = self.states[0]
    elif ia16020.value ==
        self.tele.ia16020[2]:
        #ia16020_CB_POSITIVE_NO_TEXT
      self.st = self.states[0]
    elif ia16020.value ==
        self.tele.ia16020[3]:
        #ia16020_CB_INFO_TEXT
      self.st = self.states[2]
      popup.set(True)
    elif ia16020.value ==
        self.tele.ia16020[4]:
        #ia16020_CB_POSITIVE_WITH_TEXT
      self.st = self.states[4]
    else:
       pass
  elif self.st == self.states[3]:
    if ia16020.value ==
        self.tele.ia16020[4]:
        #ia16020_CB_POSITIVE_WITH_TEXT
      self.st = self.states[4]

  elif self.st == self.states[2]:
    self.st = self.states[0]
=}
```

Listing 3.3. Source code states History 2/4

```
reaction(user_reaction) -> ia16530
    {=
  if self.st == self.states[2]:
    if user_reaction.value ==
        self.tele.user_reaction[3]:
        # USER_REACTION_POSITIVE
      ia16530.set(
      self.tele.user_reaction[3])
      self.st = self.states[3]
    elif user_reaction.value ==
        self.tele.user_reaction[4]:
        #USER_REACTION_NEGATIVE
      ia16530.set(
      self.tele.user_reaction[4])
      self.st = self.states[0]
    else:
       pass
  elif self.st == self.states[4]:
    if user_reaction.value ==
        self.tele.user_reaction[0]:
        #ok
      self.st = self.states[0]
=}
```

Listing 3.4. Source code states History 3/4

```
reaction(ia16030) -> data {=
  if self.st == self.states[4]:
    data.set(ia16030.value)
    self.st = self.states[0]
=}

reaction(cancel) -> ia16530 {=
  ia16530.set(self.tele.user_reaction[1])
  self.st = self.states[0]
=}
}
```

Listing 3.5. Source code states History 4/4

In the reactor Simu on the other hand, there are four reactions but only two inputs. Reaction two and three simply react in a comparable fashion to the inputs depending on the current state as in History. The connection between input port ia1650 and reaction for exists not because the telegram is the trigger. If the reaction is triggered by the logical action it uses the current value of the telegram, without reacting explicitly to it. The check for a valid time range and appropriate size of the request, that required two different states in SimuHistoryStateMachine, is now handled by reaction three entirely. For a valid request, it schedules the logical action send_docu_-messages at the next microstep. This triggers the fourth reaction, resulting in sending the requested data to the reactor History. A logical action is necessary, because the transition in SimuHistoryStateMachine depends on the outcome of the validity checks an not an a specific event. Reaction two reacts to the user response of the popup and can therefore schedule the same logical action, which is consistent with the behavior of SimuHistoryStateMachine if it receives a positive user response while being in the

AwaitUserReply state, transitioning into the SendDocuMessages state.

There is still the question on how the timeout is handled if the requested data is too large. If the evaluation in reaction three results in a required response from a user, it sets up the physical action time_out. A physical action is appropriate in this case because the response of the user is required in physical time. Moreover, a logical action might never trigger a response as long as the system waits for the user, since the logical time is not advancing at all during that period. After the specified time, time_out triggers reaction one, resulting in reactions in all downstream connections, and eventually a transition for both reactors back to the Idle state. Consider the physical action scheduled at tag t and the delay D. The delay is in this implementation simply the time in which we what the user to respond. Reaction one is then triggered at $T = t + D$. If the user responds in time, reaction one is still triggered at T but has no real effect because the response caused reaction two to change the state that results in no behavior responding to the timeout. Considering a user request that finished with displaying the requested date, but required confirmation because the amount of data was too large. The time out was scheduled for $T = t + D$. If the user makes yet another request, requiring a response to the popup another action is scheduled for $T^* = t^* + D$, where $t < t^*$ and $t^* < T$ because the prior action has not triggered reaction one yet. In that scenario, because there is no differentiation between both actions and reaction one is triggered before the associated timeout is supposed to have an effect as shown in Figure 3.6. This leads to unintended and in more safety critical scenarios to dangerous behavior.

To address this issue, the reactor Simu contains a state variable timeout_counter which is initialized at zero. If the physical action is scheduled the counter is increased by one. As reaction one is triggered, the counter is increased by one, with no regards of the rest of the reaction. The reaction body that is required to be invoked in the case of an valid time out, checks first if the timeout counter is not greater than one. In the example where two actions are scheduled, the counter would have the value two when the first action triggers the reaction. Except for the decrease of the value nothing would happen the first time. If the user does not respond in time, reaction one is triggered again, meeting the condition of timeout_counter $\leqslant 1$, which leads to the desired behavior of the system.

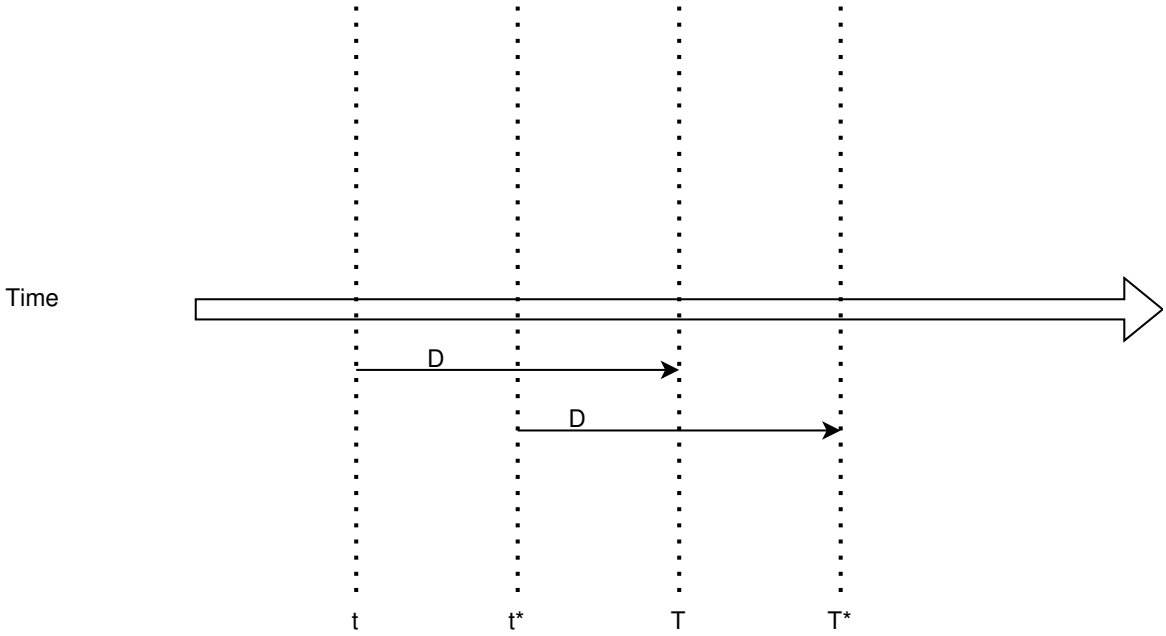

Figure 3.6. Scheduled actions on a timeline

```
reactor Simu {
  preamble {=
    import telegrams
    import db
    tele = telegrams.Telegrams()
    states =
        ["Idle","AwaitUserReaction",
        "SendDocuMessages"]
  =}
  input ia16520
  input ia16530
  output ia16020
  output ia16030
  output timeout
  state st = "Idle"
  state timeout_counter = 0

  logical action send_docu_messages
  physical action time_out
```

Listing 3.6. Source code states Simu 1/4

```
  reaction(time_out) -> ia16020,
      timeout {=
    if self.st == self.states[1] and
        (self.timeout_counter <= 1):
      ia16020.set(self.tele.ia16020[6])
          #ia16020_CB_TIMEOUT
      timeout.set(True)
      self.st = self.states[0]
    self.timeout_counter -= 1
  =}

  reaction(ia16530) -> ia16020,
      send_docu_messages {=
    if ia16530.value ==
        self.tele.user_reaction[4]:
        #USER_REACTION_NEGATIVE
      self.st = self.states[0]

    elif ia16530.value ==
        self.tele.user_reaction[3]:
        #USER_REACTION_POSITIVE
      ia16020.set(self.tele.ia16020[4])
          #ia16020_CB_POSITIVE_WITH_TEXT
      self.st = self.states[2]

      send_docu_messages.schedule(0)
  =}
```

Listing 3.7. Source code states Simu 2/4

```
reaction(ia16520) -> ia16020,
    time_out, send_docu_messages {=
  if self.st == self.states[0]:

    timerange = ia16520.value[0] <
        ia16520.value[1]
    mc = True
    #check time range
    if not timerange:
       ia16020.set(self.tele.ia16020[1])
           #ia16020_CB_NEGATIVE
       self.st = self.states[0]
       mc = False

    if mc:
       #check message count
       st,end = ia16520.value[0],
           ia16520.value[1]
       leng =
           self.db.retrieve_logs(st,
           end, size=True)
       if leng < 1:
          ia16020.set(
          self.tele.ia16020[2])
              #ia16020_CB_POSITIVE_NO_TEXT
          self.st = self.states[0]
       elif leng > 20:
          ia16020.set(
          self.tele.ia16020[3])
              #ia16020_CB_INFO_TEXT
          self.st = self.states[1]
          time_out.schedule(SEC(10))
          self.timeout_counter += 1
       else:
          ia16020.set(
          self.tele.ia16020[4])
              #ia16020_CB_POSITIVE_WITH_TEXT
          self.st = self.states[2]
          send_docu_messages.schedule(0)
=}
```

Listing 3.8. Source code states Simu 3/4

```
reaction(send_docu_messages)
    ia16520 -> ia16030 {=
  if self.st == self.states[2]:
    st,end = ia16520.value[0],
        ia16520.value[1]
    data =self.db.retrieve_logs(st,
        end)
    ia16030.set(data)
    self.st = self.states[0]
 =}
}
```

Listing 3.9. Source code states Simu 4/4

3.3 Implementation. Reactors & State Variables

In this implementation the states from SimuHistoryStateMachine and HistoryState Machine are depicted as separate composed reactors. There are multiple considerations that need to be addressed to work within the same top level architecture as the implementation before. The challenge is to relay incoming messages at the ports of the top level reactor to the appropriate composed reactor. More precisely, relaying the incoming message depending on the current state of the top level reactor to the composed reactor that depicts that state. An approach for a system like that is shown in Figure 3.7 and Figure 3.8. This solution contains an additional reactor dispatcher_in, in each the History and the Simu component. The current state of the reactor is realized inside the dispatcher_in component, since state variables are not shared with other reactors. Considering the relay function works as expected, the dispatcher_in reactor needs to be informed from the other reactors if it should change its state and what that state is. This requires a number of additional input ports at the dispatcher_in reactor and a reaction (reaction one at History) that exist only for the purpose of changing the state. Summarized, the purpose of dispatcher_in in both top level reactors is purely to change the state and relay the inputs. The same could be achieved by one dedicated reaction at the top level reactor relaying messages and changing the state. A challenge is not only the vast amount of trigger arguments for that reaction but also the fact that it can only react to one input at tag t that may result, if not handled with caution, in unintended behavior or the system not advancing correctly with respect to its logical time.

The handling of input messages at the other composed reactors is similar to the state-only implementation, but without an additional check for the current state, since this has been handled downstream.In both reactors there is an extra reaction collecting multiple outputs before forwarding them to the output port of the top level reactor. For History this is necessary since both, the Start and AwaitUserReaction reactor can send messages to the popup port. Since this can happen simultaneously, meaning both reactors write at the popup port at the same tag t leading to non-deterministic behavior, it needs a reaction to administrate that. Otherwise the LF would through an error at compilation/code-genration. For the same reason the Simu reactor handles incoming messages, that are intended for the output port ia16020. The aforementioned challenge with the physical action as a timer is solved in the exact same way as it is done in the state-only implementation.

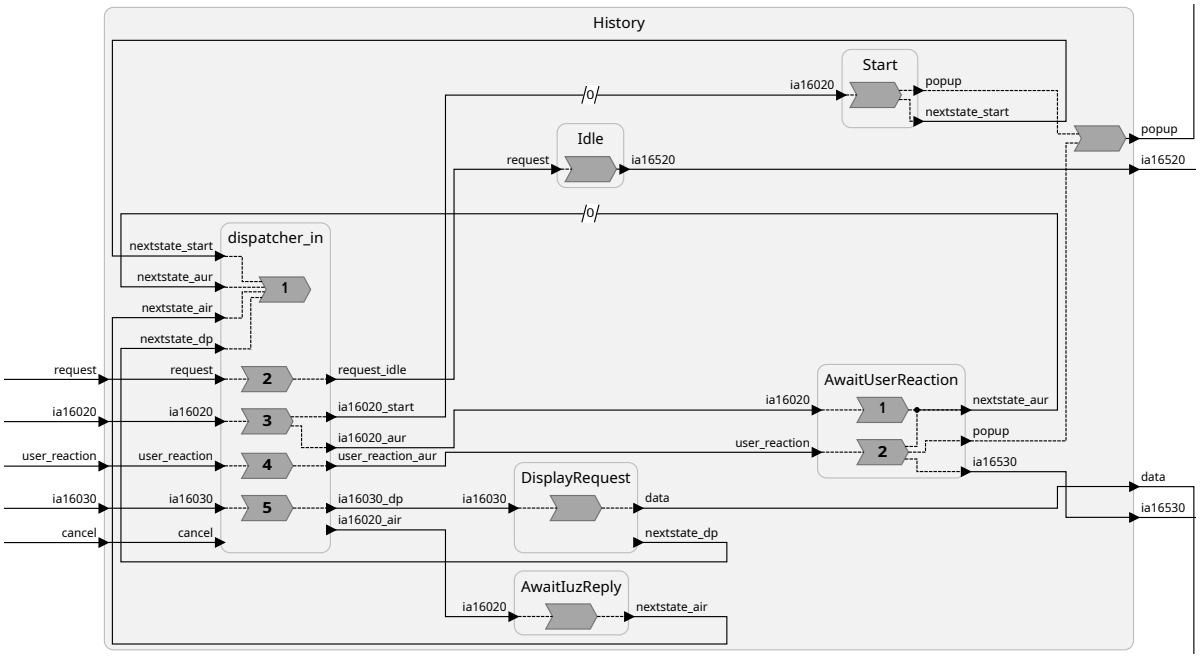

Figure 3.7. Expanded History reactor of reactor approach

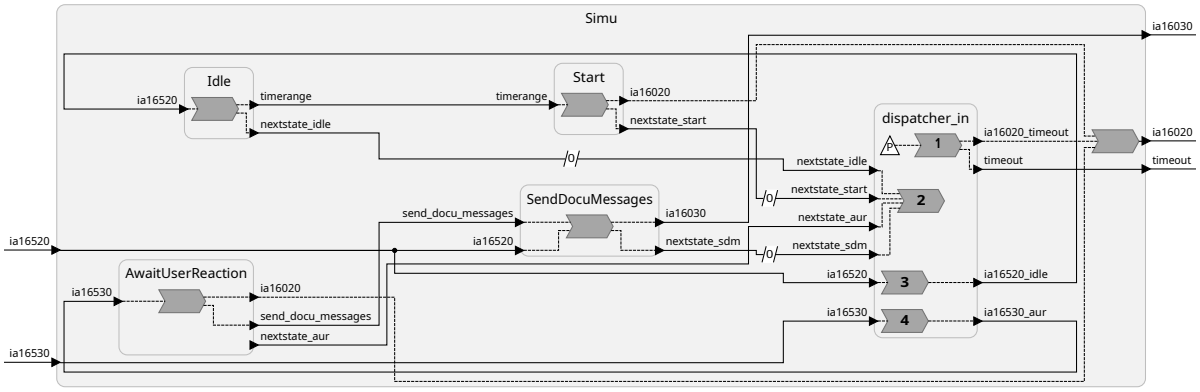

Figure 3.8. Expanded Simu reactor of reactor approach

3.4 Implementation: Modes

The last implementation does not use any state variables at all. The state driven behavior is mostly realized with the use of modal rectors and some minor additions. This is the only implementation that actually realizes the composition of the states ValidateRequest and ProcessingRequest from the SCCharts implementation. In Chapter 2

was established that modes can not be directly composed with other modes. For that reason, the reactor History and the reactor Simu contain each another composed reactor that then contains the composed states or in this case modes as it can be observed in Figure 3.9 and Figure 3.10. Transitions between modes are realized inside the reactions, depending on the current mode and the content of the received telegrams.

In order for any composed mode to be active, its top level mode needs to be active as well. This meas that any nested mode is suspended if the top level mode is suspended as well [SHL+23]. As a consequence of that, there is no need to take care of the suspension of the individual composed modes. For example, in History any mode inside the Validator reactor is suspended as soon as ValidateRequest transitions back into the Idle mode.

On a conceptual level it is straight forward how both reactors work with respect to their SCChart counterparts, yet a few changes have been made for appropriate functionality. History contains inside of the mode ValedateRequest a reaction with the number label two. This reaction only initiates the transition from ValidateRequest to the Idle mode whenever it receives a message at the done port, since the transition can not be triggered from a nested mode such as AwaitUserReaction. This is the only additional port in comparison to the state-only implementation. A almost identical approach is used in Simu, where reaction three and four both handle the transition from ProcessingRequest to the Idle mode.

Another component of History is the composed Cancel reactor. This reactor takes a cancel message directly from the user interface as it was done at HistoryStateMachine if the user cancels the request or the user application is restarted during the processing of the message. The other input port only handles a message form Validator, if the internal mode is AwaitUserReaction and the user responds either positive or negative to the popup request. A reaction inside the Cancel reactor invokes different code depending on the user reaction. In case of a positive response it simply relays the ia16530 telegram to the ia16530 output port of History, containing the information of a positive response for the Simu reactor. In case of a negative response, it relays the ia16530 telegram the same way but also triggers reaction two of History through the done output port, in order to transition the whole reactor into the Idle mode. To achieve the same functionality the reactor Cancel is not necessary and can be supplemented by a dedicated reaction inside the Validator reactor.

If History receives a request message, while being in the Idle mode, reaction one triggers the transition to ValidateRequest and forwards the content of the request message to the ia16520 output port. This is similar to the behavior of the original system as well as the unexplained functionality of the History reactor.

Consider the Simu reactor receiving a ia16520 telegram while being in the Idle state. Reaction one only changes the current mode to ProcessingRequest. But in SimuHistoryS-

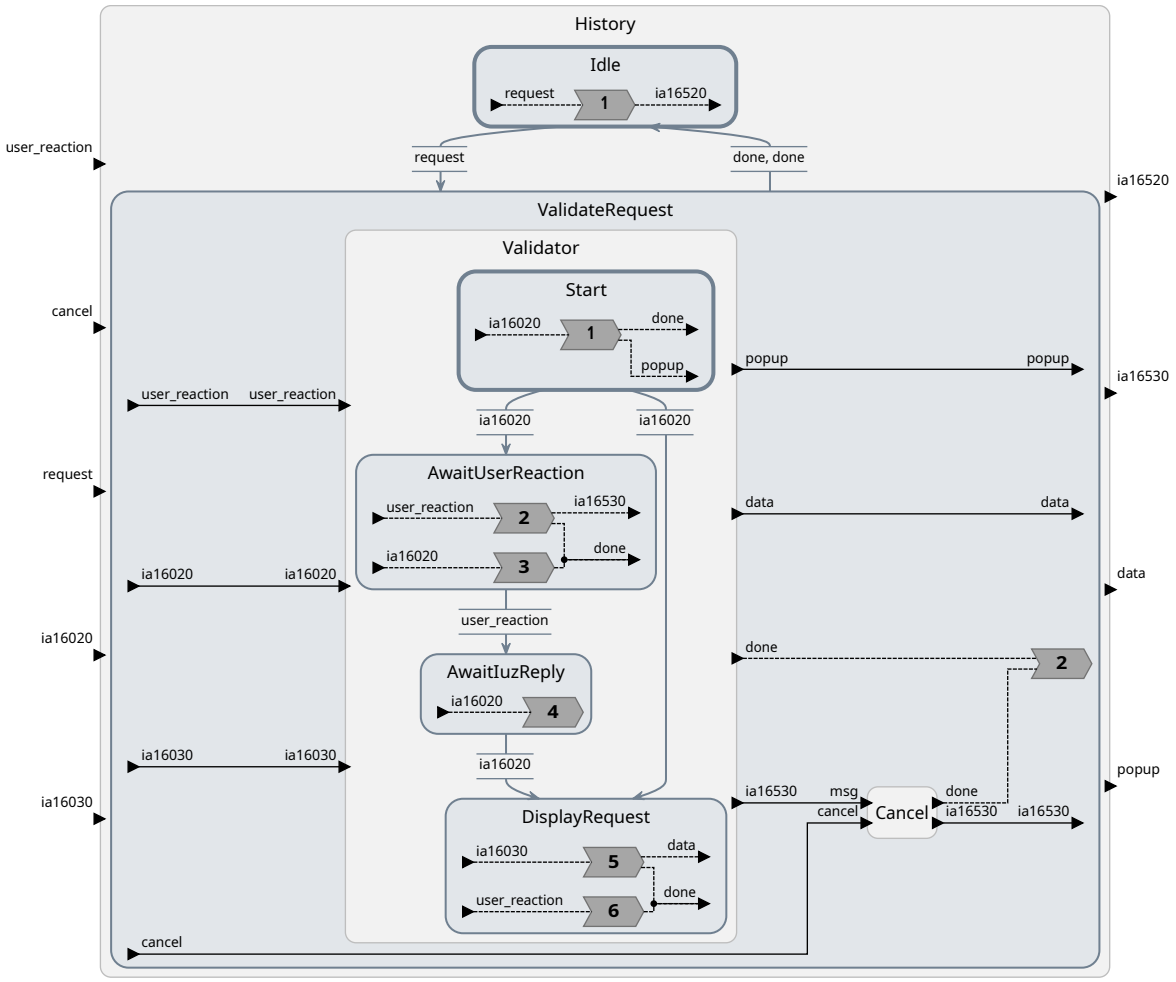

Figure 3.9. Expanded History reactor of mode approach

tateMachine the Start state and the CheckMessageCount have no event that triggers the processing of the request. For that reason both states are merged into the Processing mode of Simu. Reaction two is triggered every time after a transition into the ProcessingRequest using the keyword reset as reaction argument, forwarding the content of telegram ia16520 to the reactor RequestProcessor in order to start the validity checks for the request. In SimuHistoryStateMachine, there is also no event that triggers the process of sending the requested data date to HistoryStateMachine but the transition itself. With the use of the keyword reset as argument for reaction five, that behavior is realized in this LF implementation.

In that same fashion, every time the mode AwaitUserReaction is entered, reaction two schedules a physical action, that results in a timeout if the user does not respond in time, as seen in the original system and the other two implementations. The

challenge in both prior implementations was to ensure that an older action does not result in a premature abort of the whole process. This was done, by initializing a state variable that displayed the current amount of active timeout actions. Using modal reactors this is no longer necessary. If the user responds in time, the downstream reactions result in a transition into the Idle mode eventually, discarding the state of ProcessingRequest including the states of all nested modes and therefore discarding the scheduled physical action. In fact, the state of the mode AwaitUserReaction is discarded as soon as it is suspended for example by the transition to SendDocuMessages.

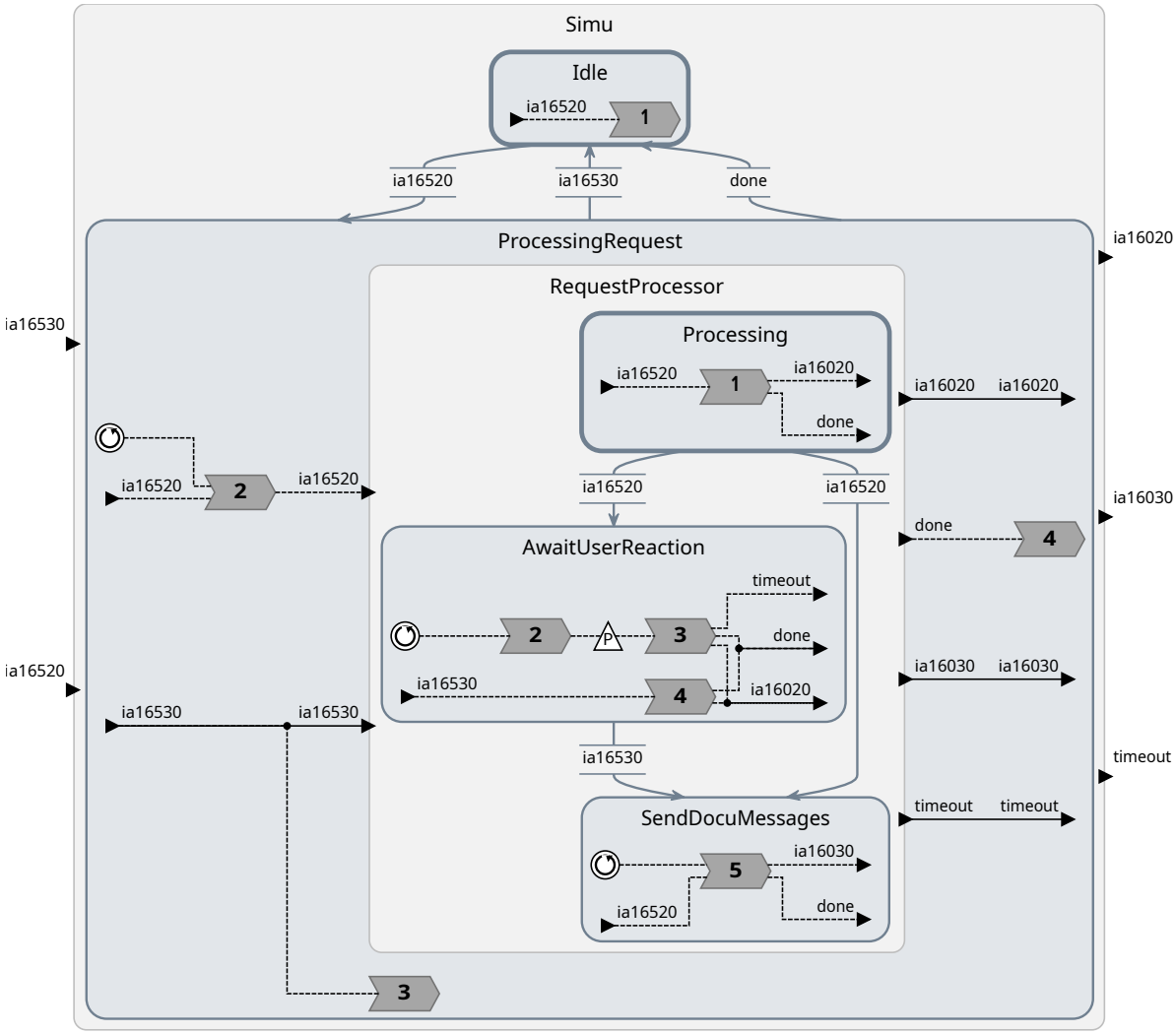

Figure 3.10. Expanded Simu reactor of mode approach

Chapter 4

Evaluation

This chapter evaluates the three different implementations from the authors perspective, with regards the translational process from SCChart notation into a LF system with special focus on complexity, noted patterns and validity. Developers from S&B and a researcher from Kiel University completed a survey after working with and browsing through the LF generated diagrams of all three different implementations. The results of that survey with regards to the utility of those diagrams as an additional layer of documentation are subject to the evaluation of the second section. Lastly, a brief introduction to the CAL Theorem, its implications for the use of LF, and the possible applications at systems of S&B are discussed.

4.1 Evaluating the Implementation

To develop a system in LF from the SCChart notation, the state-only design was the most straightforward to implement of all three. In both reactors there needed to be at least one reaction dedicated to an input port. To keep track of the state the reactor is currently in, one state variable needed to be initiated (one per reactor). From there a reaction triggered my a specific input port only needed to evaluate the current state in order to invoke the appropriate part of the reaction's body. In the Simu reactor, there were two additional reactions to the two required for both input ports. The reaction invoking an abort of the system if the user did not responded in time is negligible, because it has no relation to the state behavior. On the other hand a reaction was necessary to invoke the body of the reaction after the transition to SendDocuMessages with the use of a logical action. The same approach is required if the "empty" states are meant to be rebuild as well. Apart from the reactor abstraction and the connections between them, this was similar to common approaches if building systems with state behavior using only the essentials of a given programming language. Depending on the programming language one may not necessarily use a standard variable to keep track of the state of the system. For example the use of enums or objects in object oriented programming languages is a sufficient choice in many cases. Since reactors do not share state variables among each other, a satisfactory level of encapsulation

of a reactor was achieved, that might be more convenient in other systems, than the one that was build for this thesis. Connecting ports was also straightforward, which simplified the process of connecting the reactors and building an concurrent system with two stand alone components. Since the implementation was the least complex one, it was intuitive to reason about bugs and errors.

Unfortunately, the generated diagram obfuscated most of the state driven behavior of the system since only the reactions with their respective triggers were visible. Reasoning about the system without inspecting the underlying source code is challenging since state variables are not displayed in the generated diagrams. Moreover, the source code itself is less intuitive, because it relies on many nested conditional statements. While this still manageable for this system it will become more challenging to reason about for more complex systems. Since there was no automated approach for discarding the logical action, this was the most challenging part that resulted in several bugs during then implementation.

The second approach was the least intuitive one of all three design choices. For each depicted state, one reactor was created. To manage the change of the states and relaying the messages to the appropriate reactor another component is required, that was in this case another reactor. Ensuring that the state is changed all composed reactors required a distinct connection to the dispatcher. For fulfill LF's language constrains a separate reaction was implemented in each top level reactor, to ensure that only one connection writes to a single output port at tag t. The reactions complexity inside the composed reactors was slightly less complex than for the state-only approach. The amount of additional connections, reactors and components led to a complex and convoluted system. During development the system did often not work as expected due to cyclic dependencies which were challenging to reason about. In the attempt to resolve those dependencies, delays sometimes resulted in the system not advancing the internal logical time and therefore not working as expected. The generated diagram offers more information of the underlying system but the amount connections add to am ambiguous diagram.

Using modal reactors was predominantly consistent with the SCChart notation and and the most intuitive design choice. For each state that a separate mode was created. To nest modes within modes, the only requirement was to create an additional reactor within a mode for each layer of convolution (in this case one). Transitioning between modes was straight forward and reproducible. Invoking functionality after entering a mode, was clear-cut due to the use of the keyword **reset** as reaction argument. This can also be used for the "empty" states if desired. Discarding states after mode changes, added a layer of security especially in the case of triggering the timeout through a physical actions. That code is only invoked if the respective mode is active, made it coherent to reason about the system during development and debugging.

The additional reactor Cancel is not required and can therefore not be considered as a pattern for building a system in LF from SCChart notation. Of all three implementations, the mode design offered the most information of the underlying system from the generated diagrams. Except for the actual behavior of the functionality it provided a similar layout to the SCChart diagrams. Although, the source code was the second largest of all three implementations due to reactions, reacting to the same input ports for different modes, the code itself was coherent and modular without adding a significant amount of convolution.

4.2 Opinions from Developers

The conducted survey is not meant to offer any statistically significant results. The analysis of the diagrams was done with the intend to offer qualitative insights from people familiar with SCChart and the rebuild system from S&B. All consulted developers had no experience with LF with the exception of the person consulted from Kiel university.

When the interviewees were asked which of the three diagrams they prefer with respect to the information they provide in relation the the SCChart notation, the answer was always the mode design. It offers the most similarities to the familiar state based representation and depicted coherently which messages are important at what state. The state-only approach was the least preferred, because it obfuscated almost all internal functionality even if the representation was clear-cut. On the other hand the composed reactor implementation provided more information about the internal behavior but was convoluted and ambiguous. The vast amount of additional connections appeared clunky. After establishing the preferred design the questions aimed at a more in depth analysis. In the introduction it was mentioned that the challenge with the SCChart notation is, that it hardly provides insights about the interaction of both components. All interviewees considered the top level outline with the represented connections, a improvement to the contemporary documentation. Overall, the transitions between modes are coherent and the data flow comprehensible but the internal logic is sometimes ambiguous. One developer noted that it would be advantageous to have direct paths from the connections of a mode to its top level reactor. For example that the ia16020 connection from the mode ProcessingRequest has a direct visual connection to the ia16020 output port of Simu. Another note is that it would help in terms of documentation, if the reactions could be named as well to provide more information than just the order they are invoked at simultaneously occurring messages. Moreover, it would be advantageous to wrap existing SCChart into LF reactors, rendering the translation into modal reactors unnecessary,

while providing an improved understanding of the connections between separate but concurrently working components. The last job for the interviewees was to assess if the preferred diagram type, or under what circumstances a LF diagram, would be an improvement as an additional layer of documentation. Even if modal reactors provided a similar layout as SCChart and the interactions between different components were more comprehensible, it does not add sufficient benefits when considering the additional effort for rebuilding the system with LF. However, if existing SCChart could be wrapped inside LF reactors, the developers would likely consider it a significant improvement due to the streamlined integration and enhanced comprehensibility of system interactions under this configuration.

4.3 The CAL Theorem

In this section another example derived from a system developed at S&B, introduced in Chapter 3 and sketched in Figure 3.1, is used to briefly showcase how the CAL theorem in combination with LF can be used to build reliable distributed systems based on assumptions about the network conditions. It's crucial to note that the depicted functionality and challenges associated with S&B's system are highly simplified for illustrative purposes, and do not reflect any actual product or security standards upheld by S&B. In reality, the systems developed by SB adhere to the highest standards of security, showcasing the company's strong commitment to ensuring robust safety and reliability. The simplification here is solely for pedagogical clarity and should not be interpreted as a reflection on the rigorous professionalism and stringent security protocols maintained by S&B. The mathematical formalism in this section is reduced to the necessary minimum, but the interested reader can find a precise exposition at [LBL+21] and [LBL+23a]. In [LBL+23a] the CAL theorem is derivated by providing a series of precise definitions within the notion of traces, that are depicted as a number of sequential processes where every process is an unbounded sequence of tagged events. The precision and formalism is obfuscated in this thesis, since only a conceptual understanding with minimal mathematical formalism is required in the practical section.

The CAP theorem by Brewer states, that in a distributed system either availability (A) or consistency must be sacrificed in the presence of network partitioning. Broadly speaking, network partitioning means, that the connection between nodes is interrupted. Consistency is an agreement between nodes about a shared state and availability means that the node is able to respond to (user) input [Bre12] [Bre00].

The CAL theorem quantifies availability, consistency and apparent latency (L), thereby providing a systematic approach to understanding and analyzing the trade-

offs among these essential aspects in a distributed system. The theorem manifests as a linear system of equations within a max-plus algebra, where the structure of the equations depict the communication topology of the application. This allows compact modeling of heterogeneous networks where latencies between pairs of nodes may vary considerably [LBL+23a].

To the introduced notion of logical time and tags ($t = (a, b)$) is now extended. Let's say $\mathcal{T}$ is the totally ordered set of tags and $\mathbb{T}$ the set of all possible measurements of physical time. $\mathfrak{T} : \mathcal{T} \rightarrow \mathbb{T}$ is a monotonically nondecreasing function that provides a mapping for any tag to a physical timestamp in LF with $\mathfrak{T}(t = (a, b)) = a$. For a shared variable x in a distributed system, let's distinguish between write and read events in a number of sequential processes. In order to define *Inconsistency* consider for each write event on process j with tag t_j the related accept event on process i with tag t_i. An accept event is simply the event in a process that notes the change on x by a write event in another process. Given that, *Inconsistency* is defined as $C_{ij} = \max(\mathfrak{T}(\mathfrak{t}_\mathfrak{i} - \mathfrak{T}(t_j))$. Bounded inconsistency is called eventual consistency whereas $C_{ij} = 0$ means strong consistency [LBL+23a]. In simple terms inconsistency measures the time it takes for processes to agree over the state of a shared variable by maximizing over all write events on process j.

On the other hand, for each read event on process i with the tag t_i and T_i as the physical time of processing this event, let the *Unavailability* at the process i be $A_i = \max(T_i - \mathfrak{T}(t_i))$. With this maximization, unavailability is the measure of time taken between the time of a request and the response by the system [LBL+23a].

The processing offset O_i is defined in [LBL+23a] as the time required to process an write event. In practical terms unavailability concerns the delay from when a user request is made until the system begins processing that request. In contrast to that the processing request reflects the delay form when an external input triggers a write event until the system actually processes the write event. Apparent latency ($\mathfrak{L}_{ij}$) is defined as the maximum difference between the time a write event is processed in one node (process j) and the time the corresponding accept event is processed in another node (process i). It's termed as "apparent" because the actual latency could be affected by the synchronization differences between the local clocks of process i and process j. In fact, $\mathfrak{L}_{ij}$ is the sum of X_{ij} (execution time overhead - the time taken on node j to prepare and send the message to node i), L_{ij} (Network latency - the time taken for the message to travel across the network from node j to node i), E_{ij} (clock synchronization error - the error due to imperfect synchronization between the clocks of node i and node j) and finally O_j (processing offset - the time delay in processing the write event on node j). From that and with a given trace, the CAL theorem states

that the unavailability at process i is

$$A_i = \max\left(O_i, \max_{j \in \mathbb{N}} \left(\mathfrak{L}_{ij} - C_{ij}\right)\right)$$

in the worst case [LBL+23a]. With that relationship it is now possible to trade of availability against consistency (or vice versa) in an actual system with assumptions about the apparent latency.

Figure 3.1 depicts a system where an operator can control connected components from the iBP. One component is the ZSB2000, that is a complex and multifunctional interlocking component on railway track sections. Due to safety critical requirements the communication between the iBP and interlocking systems is relayed through a number of redundancy components to assure consistency over shared states. For simplicity reasons, the running example is represented by two components, a user interface for the operator (similar to the iBP) and a interlocking component with a signal box for a dedicated track section. Both components are connected directly and do not include any checks for validity.

Consider a train track that is divided into different train sections. Each section has its own interlocking component with an integrated signal box. Sensors in the interlocking component detect if a track section is either free, blocked by a train or faulty for an unknown reason and the signal box has a control light that can be green or red, similar to a traffic light. The simplified version of the iBP for this example is called Operating Station (OS). The screen of the OS displays the current state of the connected interlocking component (free, blocked, faulty) and the state of the signal box (green or red). A state change appears if the sensor detects a free track section, a train on the track section or an error, which is then broadcasted to all connected OSs. Provided with that information the operator can initiate a change at the signal box (switching between green and red). A switch to green can only be realized if the track section is free. This is obvious because any other behavior could lead to trains colliding on the track section. An error does not provide information if a train is on the track section or not. It simply states that the interlocking component is not able to determine if the track is blocked or free. On changing the state to either blocked or error the signal box switches automatically to the red light, to indicate to any approaching train that the next section can not be entered. A green light indicates that the next section can be entered. Switching the signal box' light to red, does not have any contains related to the state of the interlocking component. The operator must always be able to initiate a change of the signal box towards red. This is because unlike switching to green, the worst case would be a train waiting on the track.

Imagine a track section where construction is being undertaken. This could change the state to error, but we assume that the sensor detecting a train on the tracks still

works. It is not uncommon in those situations that the construction work is interrupted several times a day for a passing train to not discontinue the entire train service. Since the operator would be informed about that, they must be able to change the light of the signal box to green as long as the state of the signal box is only error and not blocked. This would require something resembling a super-user-request. A system representing the described example build in LF is displayed in Figure 4.1.

The IBP reactor contains three physical actions that can trigger reaction one, two and five. In a real system those physical actions are asynchronous inputs from the user at the OS initiating changes in the state of the interlocking components. Both reactors contain the local state variable section, which depicts the state of the interlocking component and the local state variable signal, that represents the state of the signal box. Reaction five send a message over the signal port to the component TrackSection in order to change the state of the signal variable to green (change the light of the signal box). If the state of TrackSection is free, it changes the state of signal and broadcasts both state variables to all connected OSs (in this example only one). If the intended change was red, TrackSection changes the state signal to red with no regards to the current state of section and broadcasts the updated state variables. Reaction four of IBP changes the local state variables to the values received from the TrackSection components. In TrackSection, reactions three, four and five change the local state of signal and section with regards to the dedicated sensor data and transmits the changed state values to the IBP component. For example, physical action block triggers a reaction that changes section to block and signal to red. The operator can initiate a super user query, which resulting in the invocation of the downstream reaction one of TrackSection which sends a message containing the current state of section over the su_answer port to the IBP reactor. This changes the state variable su_mode of IBP to change to True if, and only if the message value was error. From there the operator can trigger reaction two, that results downstream in TrackSection in a change of signal to green even if the state of section in TrackSection is not free. This represents the aforementioned behavior, where the operator can initiate a switch to green, even if the track section is actually not free.

To integrate the results from the CAL theorem, only the transmission direction from TrackSection towards IBP is considered to trade off availability for consistency. Consider the connection between the output port publish and the input port su_answer to suffer from network latency. This means that the state of both variables is different at TrackSection from the state at IBP. In practice, an operator could see a different state of the track section and initiating a change of the signal box' state considering this false information. With a regular connection in LF this could not happen, because if the state change in TrackSection happens "before" the initiation by the operator, the system would recognize that $t_{trackSection} \leqslant t_{IBP}$. We therefor would have strong

consistency resulting in the operator not being able to send the request before there is an agreement over the state in both systems. With network latency this may result in unavailability, meaning the operator can not interact with the signal box until consistency is achieved. Since the operator should be able to always request a change to red this behavior is not desired. Furthermore, even if the state of section is not free in TrackSection but the operator sees an outdated state, resulting in an attempt to change the signal box to green, this would not cause any problems since that case is handled as seen in line 14-22 of Listing 4.4. The CAL theorem can be used to relay consistency by a specified amount allowing to account for network latency and enabling the operator to be able to intact with the system even if the states are not consistent at the time. For this example the apparent latency is simplified to only depend on network latency: $\mathfrak{L}_{ij} = L_{ij}$. It is also assumed that the processing offset is $O_{ij} = 0$ resulting in $A_i = \max\left(0, \max_{j \in \mathbb{N}}\left(L_{ij} - C_{ij}\right)\right)$ where $C = 500ms$ to allow for a assumed network latency of 500 ms. In LF this is achieved by using the **after** keyword applying a logical delay to the connection (line 34 in Listing 4.4). What this means that a logical delay between the initiation of the change of section in TrackSection and the recording of that update in IBP, allows for availability at the OS even if the system is not consistent. It is important to note, that the availability is bounded by 500 ms. How a violation of that assumption can be addressed is discussed in [LBL+23a]. Even with the logical delay, this design assures eventual consistency. The super user request on the other hand requires strong consistency. An operator should only be able to change the signal box' light to green, if the value of section is the same across all nodes. In this design, this is achieved on default since the involved reactions logically depend on each other making it impossible for the operator to change the state of signal before the system is consistent with regards to the state variable of section. Note that with high latency, this results in unavailability for the operator.

In this section the CAL theorem was used in combination with LF to showcase how to trade off availability against consistency in a rather simple example. The findings of [LBL+23a] and [LAB+23] offer a much more detailed approach, where more precise considerations of apparent latency are applied. Moreover, they show how the decision between centralized and decentralized execution in LF can be leveraged for more flexibility considering availability and consistency. They provide insights on how to handle situations where the assumptions made about the network conditions fail, which was not discussed in this chapter.

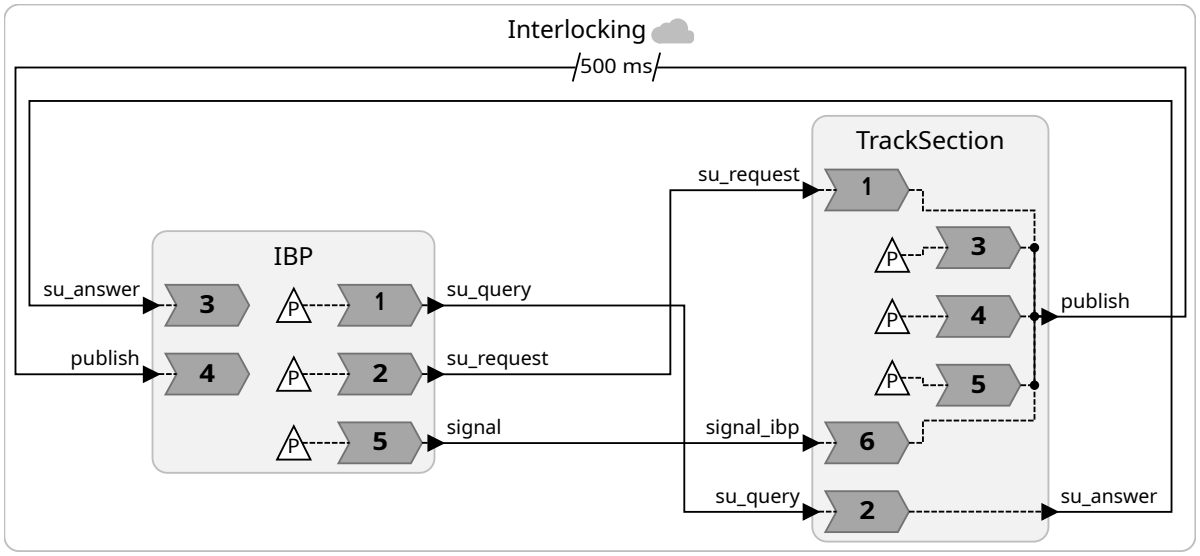

Figure 4.1. Diagram of interlocking and OS component in LF

```
reactor IBP {
  preamble {=
    track_states = ["free", "error",
        "blocked"]
    signal_states = ["red","green"]
  =}

  input publish
  input su_answer
  output signal
  output su_query
  output su_request
  state section
  state signal
  state su_mode = False
  physical action user_input
  physical action su_qu
  physical action su_re

  reaction(su_qu) -> su_query {=
    su_query.set()
  =}

  reaction(su_re) -> su_request {=
    if self.su_mode:
       su_request.set(True)
    else:
       su_request.set(False)

    self.su_mode = False
  =}
```

Listing 4.1. Source code of interlocking and OS component in LF 1/4

```
  reaction(su_answer) {=
    if su_answer.value == "error":
       self.su_mode = True
  =}

  reaction(publish) {=
    self.section = publish.value[0]
    self.signal = publish.value[1]
  =}

  reaction(user_input) -> signal {=
    if user_input.value:# True mean
        set the signal to green
       if self.section == "free":
          signal.set(True)
       else:
          pass
    else:
       signal.set(False)
  =}
}
```

Listing 4.2. Source code of interlocking and OS component in LF 2/4

```
reactor TrackSection {
  preamble {=
    track_states = ["free", "error",
        "blocked"]
    signal_states = ["red","green"]
  =}
  input signal_ibp
  input su_query
  input su_request
  output publish
  output su_answer
  state section
  state signal
  physical action block
  physical action error
  physical action free

  reaction(su_request) -> publish {=
    if su_request.value:
       self.signal =
           self.signal_states[1]
       publish.set([self.section,
           self.signal])
    else:
       publish.set([self.section,
           self.signal])
  =}

  reaction(su_query) -> su_answer {=
    su_answer.set(self.section)
  =}

  reaction(block) -> publish {=
    self.section = self.track_states[2]
    self.signal = self.signal_states[0]
    publish.set([self.section,
        self.signal])
  =}
```

Listing 4.3. Source code of interlocking and OS component in LF 3/4

```
  reaction(error) -> publish {=
    self.section = self.track_states[1]
    self.signal = self.signal_states[0]
    publish.set([self.section,
        self.signal])
  =}

  reaction(free) -> publish {=
    self.section = self.track_states[0]
    self.signal = self.signal_states[0]
    publish.set([self.section,
        self.signal])
  =}

  reaction(signal_ibp) -> publish {=
    if signal_ibp.value:
       if self.section == "free":
          self.signal =
              self.signal_states[1]
          publish.set([self.section,
              self.signal])
       else:
          publish.set([self.section,
              self.signal])
    else:
       self.signal =
           self.signal_states[0]
       publish.set([self.section,
           self.signal])
  =}
}

federated reactor Interlocking {
  ibp = new IBP()
  ts = new TrackSection()

  ibp.su_request -> ts.su_request
  ibp.signal -> ts.signal_ibp
  ibp.su_query -> ts.su_query

  ts.publish -> ibp.publish after 500
      ms
  ts.su_answer -> ibp.su_answer
}
```

Listing 4.4. Source code of interlocking and OS component in LF 4/4

Chapter 5

Conclusion & Further Work

This thesis shows how LF can be used to build working systems from existing ones that were originally modeled with SCCharts. The implementation using composed reactors did not offerer a desirable design choice due to its convoluted and ambiguous pattern. On the other hand both, the state-only and modal reactor approach follow recognizable patterns, which lead to the question if those patterns are consistent enough to derive ways in witch the translational process adapted in this thesis can be automated.

In the first approach at least one state variable, to depict every state of the SCChart is required. Behavior depending on the current state would be addressed using conditional logic in the reactions if needed. Even addressing the transition to and between "empty" states is possible by using logical actions. While this worked at one instance of this program, there is no guarantee, that it does not cause undesired behavior in more complex systems. If some functionality depends on other events or needs to run concurrently, the system could behave unexpectedly because the logical time is not advancing in a fashion that represents the functionality of the original system. Contained states (e.g. `Start` in `ValidateRequest`) could be approached by yet another state variable. But this is not straight forward in consideration of automating the translational process, because more complexity may imply specific behavior that does not follow a strict pattern.

While using the modal reactor approach, which offers advantages over the state-variable design, such as discarding or suspending local states, the challenges towards automation are similar. Even if states in SCCharts are not the same as modes in LF one mode per SCChart state is required. Integrating "empty" states and triggering behavior purely by entering another state is fairly elegant in this design by using the keyword **reset** as reaction argument. Composed states can be addressed by creating a composed reactor for every nested level. To trigger transitions of the top level mode from a nested one requires an extra reaction at the top level and and extra port at every nested mode. This was feasible for systems at that level of complexity but can become a very challenging task for every extra level of composition. Even if this design offered the most consistent patterns for the translational process, it remains unclear from this work if those patterns hold for more complex systems. Over all, all

approaches submit insights of the feasibility, of automation of the internal behavior. Since components like the discussed ones are communicating over a network, it is not obvious from the SCChart notation on how to connect them in an automated sense.

This relates to the question if the generated LF diagrams are considered an advantageous additional layer of documentation by developers at S&B. The mode design was consistently described as the most intuitive and coherent choice, with regards to displaying information of the internal logic if single components and the communication between them. Yet, rebuilding existing SCChart systems in LF for the exclusive purpose of additional documentation that of offers a better understanding on how distributed components interact with each other, is an infeasible task for S&B. This conclusion does not reflect on the value LF provides for the development of deterministic, time sensitive and reactive systems at any point. Indeed, the substantial and significant benefits of LF for that purpose have been showcased by a vast number of research contributions, from which a proportion has been used and cited for this thesis.

Since rebuilding SCChart systems in LF may not be the most considerable approach, future work may address the aspect of integrating SCCharts in LF. How the diagram representation is integrated, exceeds the authors expertise and is left open for professionals in that domain. For a functional integration any further work could explore on how the event driven behavior of SCChart can be connected to the event driven nature of LF. Furthermore, the examined SCChart systems revealed that events, such as transitions, invoked methods that triggered events in the connected component. Exploring on how those connections can be combined using import and output ports in LF appears auspicious.

When it comes to the diagrams containing modes, one developer noted that it would advance the comprehension of the system, if output ports of contained modes would visually connect in a direct way to the top level ports. If this idea is desirable, especially if many outgoing connections link to one port, is left to be decided by the LF community. Naming reactions and displaying that in the generated diagrams is worth exploring, because it would offer more insights about the internal logic and the functional purpose of reactions if named sensibly. Integrating that functionality both into the syntax and diagrams appears feasible from the authors perspective.

Section 4.3 briefly demonstrated how LF can be used to build even more secure and testable distributed systems, while adjusting the requirements for availability and consistency based on specific business decisions. Even though, the example was domain specific to RS, one could easily conclude that the insights can be adopted by other applications. Because of that, and the substantial scientific contributions made explicitly for distributed systems by the LF community, future work may consider additional features in LF based on the consequences of the CAL theorem.

Appendix A

Source Code: Modal Reactors Implementation

```
target Python {
  files: ["../external/telegrams.py", "../external/db.py", "../external/gui.py"]
}

import Simu from "Simu.lf"
import History from "History.lf"
import Gui from "Gui.lf"

main reactor HistoryStateMachine {
  simu = new Simu()
  hist = new History()
  gui = new Gui()

  hist.ia16520 -> simu.ia16520 after 0
  hist.ia16530 -> simu.ia16530 after 0

  simu.ia16020 -> hist.ia16020
  simu.ia16030 -> hist.ia16030

  gui.user_reaction -> hist.user_reaction
  gui.request -> hist.request
  gui.cancel -> hist.cancel

  hist.data -> gui.data
  hist.popup -> gui.popup
  simu.timeout -> gui.timeout
}
```

Listing A.1. Source code modes HistoryStateMachine

```
target Python {
```

```
  files: ["../external/telegrams.py"]
}

import Validator from "Validator.lf"
import Cancel from "Validator.lf"

reactor History {
  preamble {=
    import telegrams
    tele = telegrams.Telegrams()
  =}
  input cancel
  input request
  input ia16020
  input ia16030
  input user_reaction

  output ia16520
  output ia16530
  output data
  output popup

  initial mode Idle {
    reaction(request) -> ia16520, reset(ValidateRequest) {=
      ia16520.set(request.value)
      ValidateRequest.set()
    =}
  }

  mode ValidateRequest {
    validatior = new Validator()
    relais = new Cancel()
    ia16020 -> validatior.ia16020
    ia16030 -> validatior.ia16030
    user_reaction -> validatior.user_reaction

    validatior.popup -> popup
    validatior.data -> data

    cancel -> relais.cancel
    validatior.ia16530 -> relais.msg
```

```
43    relais.ia16530 -> ia16530
44    reaction(validatior.done, relais.done) -> reset(Idle) {=
45      Idle.set()
46    =}
47  }
48 }
```

Listing A.2. Source code modes History

```
1  target Python {
2    files: ["../external/telegrams.py"]
3  }
4
5  reactor Validator {
6    preamble {=
7      import telegrams
8      tele = telegrams.Telegrams()
9    =}
10   input ia16020
11   input ia16030
12   input user_reaction
13
14   output ia16530
15   output done
16   output data
17   output popup
18
19   initial mode Start {
20     reaction(ia16020) -> done, popup, reset(AwaitUserReaction),
         reset(DisplayRequest) {=
21       if ia16020.value == self.tele.ia16020[1]: #ia16020_CB_NEGATIVE
22         done.set(True)
23       elif ia16020.value == self.tele.ia16020[2]: #ia16020_CB_POSITIVE_NO_TEXT
24         done.set(True)
25       elif ia16020.value == self.tele.ia16020[3]: #ia16020_CB_INFO_TEXT
26         AwaitUserReaction.set()
27         popup.set(True)
28       elif ia16020.value == self.tele.ia16020[4]: #ia16020_CB_POSITIVE_WITH_TEXT
29         DisplayRequest.set()
30       else:
31         pass
32     =}
```

```
33  }
34
35  mode AwaitUserReaction {
36    reaction(user_reaction) -> ia16530, done, reset(AwaitIuzReply) {=
37      if user_reaction.value == self.tele.user_reaction[3]: # USER_REACTION_POSITIVE
38         ia16530.set(self.tele.user_reaction[3])
39         AwaitIuzReply.set()
40      elif user_reaction.value == self.tele.user_reaction[4]: #USER_REACTION_NEGATIVE
41         ia16530.set(self.tele.user_reaction[4])
42         done.set(True)
43
44      else:
45         pass
46    =}
47
48    reaction(ia16020) -> done {=
49      done.set(True)
50    =}
51  }
52
53  mode AwaitIuzReply {
54    reaction(ia16020) -> reset(DisplayRequest) {=
55      if ia16020.value == self.tele.ia16020[4]: #ia16020_CB_POSITIVE_WITH_TEXT
56         DisplayRequest.set()
57    =}
58  }
59
60  mode DisplayRequest {
61    reaction(ia16030) -> data, done {=
62      data.set(ia16030.value)
63      done.set(True)
64    =}
65
66    reaction(user_reaction) -> done {=
67      if user_reaction.value == self.tele.user_reaction[0]: #ok
68         done.set(True)
69    =}
70  }
71 }
72
73 reactor Cancel {
```

```
  input cancel
  input msg

  output ia16530
  output done

  reaction(cancel) -> ia16530, done {=
    ia16530.set(self.tele.user_reaction[1])
    done.set(True)
  =}

  reaction(msg) -> ia16530 {=
    ia16530.set(msg.value)
  =}
}
```

Listing A.3. Source code modes Validator

```
target Python {
  files: ["../external/telegrams.py"]
}

import RequestProcessor from "RequestProcessor.lf"

reactor Simu {
  preamble {=
    import telegrams
    tele = telegrams.Telegrams()
  =}
  input ia16520
  input ia16530
  output ia16020
  output ia16030
  output timeout

  initial mode Idle {
    reaction(ia16520) -> reset(ProcessingRequest) {=
      ProcessingRequest.set()
    =}
  }

  mode ProcessingRequest {
```

```
25    processor = new RequestProcessor()
26    ia16530 -> processor.ia16530
27    processor.ia16030 -> ia16030
28    processor.ia16020 -> ia16020
29    processor.timeout -> timeout
30    reaction(reset) ia16520 -> processor.ia16520 {=
31      processor.ia16520.set(ia16520.value)
32    =}
33
34    reaction(ia16530) -> reset(Idle) {=
35      if ia16530.value == self.tele.user_reaction[1]: #USER_REACTION_CANCEL
36         Idle.set()
37    =}
38
39    reaction(processor.done) -> reset(Idle) {=
40      Idle.set()
41    =}
42  }
43 }
```

Listing A.4. Source code modes Simu

```
1  target Python {
2    files: ["../external/telegrams.py"]
3  }
4
5  reactor RequestProcessor {
6    preamble {=
7      import telegrams
8      import db
9      tele = telegrams.Telegrams()
10   =}
11   input ia16520
12   input ia16530
13
14   output ia16020
15   output ia16030
16   output done
17   output timeout
18
19   initial mode Processing {
20     reaction(ia16520) -> reset(AwaitUserReaction), reset(SendDocuMessages), ia16020,
```

```
      done {=
21    timerange = ia16520.value[0] < ia16520.value[1]
22    mc = True
23    #check time range
24    if not timerange:
25       ia16020.set(self.tele.ia16020[1]) #ia16020_CB_NEGATIVE
26       done.set(True)
27       mc = False
28
29    if mc:
30       #check message count
31       st,end = ia16520.value[0], ia16520.value[1]
32       leng = self.db.retrieve_logs(st, end, size=True)
33       if leng < 1:
34          ia16020.set(self.tele.ia16020[2]) #ia16020_CB_POSITIVE_NO_TEXT
35          done.set(True)
36       elif leng > 20:
37          ia16020.set(self.tele.ia16020[3]) #ia16020_CB_INFO_TEXT
38          AwaitUserReaction.set()
39       else:
40          ia16020.set(self.tele.ia16020[4]) #ia16020_CB_POSITIVE_WITH_TEXT
41          SendDocuMessages.set()
42  =}
43 }
44
45 mode AwaitUserReaction {
46   physical action time_out
47   reaction(reset) -> time_out {=
48     time_out.schedule(SEC(10))
49   =}
50
51   reaction(time_out) -> done, ia16020, timeout {=
52     ia16020.set(self.tele.ia16020[6]) #ia16020_CB_TIMEOUT
53     timeout.set(True)
54     done.set(True)
55   =}
56
57   reaction(ia16530) -> done, ia16020, reset(SendDocuMessages) {=
58     if ia16530.value == self.tele.user_reaction[4]: #USER_REACTION_NEGATIVE
59        done.set(True)
60     elif ia16530.value == self.tele.user_reaction[3]: #USER_REACTION_POSITIVE
```

```
61          ia16020.set(self.tele.ia16020[4]) #ia16020_CB_POSITIVE_WITH_TEXT
62          SendDocuMessages.set()
63        =}
64      }
65
66      mode SendDocuMessages {
67        reaction(reset) ia16520 -> ia16030, done {=
68          st,end = ia16520.value[0], ia16520.value[1]
69          data =self.db.retrieve_logs(st, end)
70          ia16030.set(data)
71          done.set(True)
72        =}
73      }
74    }
```

Listing A.5. Source code modes RequestProcessor

Appendix B

Source Code: Reactors & State Variables Implementation

```
target Python {
  keepalive: true,
  files: ["../external/telegrams.py", "../external/db.py", "../external/gui.py"]
}

import Simu from "Simu.lf"
import History from "History.lf"
import Gui from "Gui.lf"

main reactor HistoryStateMachine {
  simu = new Simu()
  hist = new History()
  gui = new Gui()

  hist.ia16520 -> simu.ia16520 after 0
  hist.ia16530 -> simu.ia16530 after 0

  simu.ia16020 -> hist.ia16020
  simu.ia16030 -> hist.ia16030

  gui.user_reaction -> hist.user_reaction
  gui.request -> hist.request
  gui.cancel -> hist.cancel

  hist.data -> gui.data
  hist.popup -> gui.popup after 0
  simu.timeout -> gui.timeout
}
```

Listing B.1. Source code modes HistoryStateMachine

```
target Python {
  files: ["../external/telegrams.py"]
}

reactor Idle {
  preamble {=
    import telegrams
    tele = telegrams.Telegrams()
  =}
  input request
  output ia16520

  reaction(request) -> ia16520 {=
    ia16520.set(request.value)
  =}
}

reactor Start {
  preamble {=
    import telegrams
    tele = telegrams.Telegrams()
    states = ["Idle", "Start","AwaitUserReaction","AwaitIuzReply", "DisplayRequest"]
  =}
  input ia16020
  output popup
  output nextstate_start

  reaction(ia16020) -> popup, nextstate_start {=
    if ia16020.value == self.tele.ia16020[1]: #ia16020_CB_NEGATIVE
      nextstate_start.set(self.states[0])
    elif ia16020.value == self.tele.ia16020[2]: #ia16020_CB_POSITIVE_NO_TEXT
      nextstate_start.set(self.states[0])
    elif ia16020.value == self.tele.ia16020[3]: #ia16020_CB_INFO_TEXT
      nextstate_start.set(self.states[2])
      popup.set(True)
    elif ia16020.value == self.tele.ia16020[4]: #ia16020_CB_POSITIVE_WITH_TEXT
      nextstate_start.set(self.states[4])
    else:
       pass
  =}
```

```
}

reactor AwaitUserReaction {
  preamble {=
    import telegrams
    tele = telegrams.Telegrams()
  =}
  input ia16020
  input user_reaction
  output ia16530
  output popup
  output nextstate_aur

  reaction(ia16020) -> nextstate_aur {=
    nextstate_aur.set(self.states[0])
  =}

  reaction(user_reaction) -> popup, ia16530, nextstate_aur {=
    if user_reaction.value == self.tele.user_reaction[3]: # USER_REACTION_POSITIVE
      ia16530.set(self.tele.user_reaction[3])
      nextstate_aur.set(self.states[3])
    elif user_reaction.value == self.tele.user_reaction[4]: #USER_REACTION_NEGATIVE
      ia16530.set(self.tele.user_reaction[4])
      nextstate_aur.set(self.states[0])
    else:
       pass
  =}
}

reactor AwaitIuzReply {
  preamble {=
    import telegrams
    tele = telegrams.Telegrams()
  =}
  input ia16020
  output nextstate_air

  reaction(ia16020) -> nextstate_air {=
    nextstate_air.set(self.states[4])
  =}
}
```

```
82
83 reactor DisplayRequest {
84   preamble {=
85     import telegrams
86     tele = telegrams.Telegrams()
87   =}
88   input ia16030
89   output data
90   output nextstate_dp
91
92   reaction(ia16030) -> data {=
93     data.set(ia16030.value)
94     nextstate_dp.set(self.states[0])
95   =}
96 }
97
98 reactor dispatcher_in {
99   preamble {=
100    import telegrams
101    tele = telegrams.Telegrams()
102    states = ["Idle", "Start","AwaitUserReaction","AwaitIuzReply", "DisplayRequest"]
103  =}
104
105  input cancel
106  input request
107  input ia16020
108  input ia16030
109  input user_reaction
110  input nextstate_start
111  input nextstate_aur
112  input nextstate_air
113  input nextstate_dp
114
115  output request_idle
116  output ia16020_start
117  output ia16020_aur
118  output ia16020_air
119  output ia16030_dp
120  output user_reaction_aur
121
122  state st = "Idle"
```

```
123
124   reaction(nextstate_start, nextstate_aur, nextstate_air, nextstate_dp) {=
125     if nextstate_start.is_present:
126       self.st = nextstate_start.value
127     elif nextstate_aur.is_present:
128       self.st = nextstate_aur.value
129     elif nextstate_air.is_present:
130       self.st = nextstate_air.value
131     elif nextstate_dp.is_present:
132       self.st = nextstate_dp.value
133   =}
134
135   reaction(request) -> request_idle {=
136     if self.st == self.states[0]:
137         request_idle.set(request.value)
138         self.st = self.states[1]
139   =}
140
141   reaction(ia16020) -> ia16020_start, ia16020_aur {=
142     if self.st == self.states[1]:
143       ia16020_start.set(ia16020.value)
144     elif self.st == self.states[2]:
145       ia16020_aur.set(ia16020.value)
146   =}
147
148   reaction(user_reaction) -> user_reaction_aur {=
149     if self.st == self.states[2]:
150         user_reaction_aur.set(user_reaction.value)
151   =}
152
153   reaction(ia16030) -> ia16030_dp {=
154     if self.st == self.states[4]:
155         ia16030_dp.set(ia16030.value)
156   =}
157 }
158
159 reactor History {
160   input cancel
161   input request
162   input ia16020
163   input ia16030
```

```
164 input user_reaction
165
166 output ia16520
167 output ia16530
168 output data
169 output popup
170
171 idle = new Idle()
172 start = new Start()
173 await_user_reaction = new AwaitUserReaction()
174 await_iuz_reply = new AwaitIuzReply()
175 display_request = new DisplayRequest()
176 dis_in = new dispatcher_in()
177
178 cancel -> dis_in.cancel # dis_out = new dispatcher_out()
179 request -> dis_in.request
180 ia16020 -> dis_in.ia16020
181 ia16030 -> dis_in.ia16030
182 user_reaction -> dis_in.user_reaction
183
184 dis_in.request_idle -> idle.request
185 idle.ia16520 -> ia16520
186
187 dis_in.ia16020_start -> start.ia16020 after 0
188
189 dis_in.user_reaction_aur -> await_user_reaction.user_reaction
190 dis_in.ia16020_aur -> await_user_reaction.ia16020
191 await_user_reaction.ia16530 -> ia16530
192
193 dis_in.ia16020_air -> await_iuz_reply.ia16020
194
195 dis_in.ia16030_dp -> display_request.ia16030
196 display_request.data -> data
197
198 start.nextstate_start -> dis_in.nextstate_start
199 await_user_reaction.nextstate_aur -> dis_in.nextstate_aur after 0
200 await_iuz_reply.nextstate_air -> dis_in.nextstate_air
201 display_request.nextstate_dp -> dis_in.nextstate_dp
202
203 reaction(start.popup, await_user_reaction.popup) -> popup {=
204   if start.popup.is_present:
```

```
205       popup.set(start.popup.value)
206     elif await_user_reaction.popup.is_present:
207       popup.set(await_user_reaction.popup.value)
208   =}
209 }
```

Listing B.2. Source code Reactors & State Variables History

```
1  target Python {
2    keepalive: true,
3    files: ["../external/telegrams.py"]
4  }
5
6  reactor Idle {
7    preamble {=
8      import telegrams
9      tele = telegrams.Telegrams()
10     states = ["Idle","Start", "AwaitUserReaction", "SendDocuMessages"]
11   =}
12   input ia16520
13   output timerange
14   output nextstate_idle
15
16   reaction(ia16520) -> timerange, nextstate_idle {=
17     nextstate_idle.set(self.states[1])
18     timerange.set(ia16520.value)
19   =}
20 }
21
22 reactor Start {
23   preamble {=
24     import telegrams
25     tele = telegrams.Telegrams()
26     states = ["Idle","Start", "AwaitUserReaction", "SendDocuMessages"]
27   =}
28   input timerange
29   output ia16020
30   output nextstate_start
31
32   reaction(timerange) -> ia16020, nextstate_start {=
33     timerange = timerange.value[0] < timerange.value[1]
34     mc = True
```

```
35    #check time range
36    if not timerange:
37       ia16020.set(self.tele.ia16020[1]) #ia16020_CB_NEGATIVE
38       nextstate_idle.set(self.states[0])
39       mc = False
40
41    if mc:
42       #check message count
43       st,end = timerange.value[0], timerange.value[1]
44       leng = self.db.retrieve_logs(st, end, size=True)
45       if leng < 1:
46          ia16020.set(self.tele.ia16020[2]) #ia16020_CB_POSITIVE_NO_TEXT
47          nextstate_start.set(self.states[0])
48       elif leng > 20:
49          ia16020.set(self.tele.ia16020[3]) #ia16020_CB_INFO_TEXT
50          nextstate_start.set(self.states[2])
51       else:
52          ia16020.set(self.tele.ia16020[4]) #ia16020_CB_POSITIVE_WITH_TEXT
53          nextstate_start.set(self.states[3])
54  =}
55 }
56
57 reactor AwaitUserReaction {
58  preamble {=
59    import telegrams
60    tele = telegrams.Telegrams()
61  =}
62  input ia16530
63  output ia16020
64  output send_docu_messages
65  output nextstate_aur
66
67  reaction(ia16530) -> ia16020, send_docu_messages {=
68    if ia16530.value == self.tele.user_reaction[4]: #USER_REACTION_NEGATIVE
69      nextstate_aur.set(self.states[0])
70
71    elif ia16530.value == self.tele.user_reaction[3]: #USER_REACTION_POSITIVE
72      ia16020.set(self.tele.ia16020[4]) #ia16020_CB_POSITIVE_WITH_TEXT
73      nextstate_aur.set(self.states[2])
74      send_docu_messages.set(True)
75  =}
```

```
}

reactor SendDocuMessages {
  preamble {=
    import telegrams
    tele = telegrams.Telegrams()
  =}
  input send_docu_messages
  input ia16520
  output ia16030
  output nextstate_sdm

  reaction(send_docu_messages) ia16520 -> ia16030, nextstate_sdm {=
    st,end = ia16520.value[0], ia16520.value[1]
    data =self.db.retrieve_logs(st, end)
    ia16030.set(data)
    nextstate_sdm.set(self.states[0])
  =}
}

reactor dispatcher_in {
  preamble {=
    import telegrams
    tele = telegrams.Telegrams()
    states = ["Idle","Start", "AwaitUserReaction", "SendDocuMessages"]
  =}

  input ia16520
  input ia16530
  input nextstate_idle
  input nextstate_start
  input nextstate_aur
  input nextstate_sdm

  output ia16520_idle
  output ia16530_aur
  output ia16020_timeout
  output timeout

  state st = "Idle"

```

```
117   physical action time_out
118   state timeout_counter = 0
119
120   reaction(time_out) -> ia16020_timeout, timeout {=
121     if self.st == self.states[1] and (self.timeout_counter <= 1):
122       ia16020_timeout.set(self.tele.ia16020[6]) #ia16020_CB_TIMEOUT
123       timeout.set(True)
124       self.st = self.states[0]
125     self.timeout_counter -= 1
126   =}
127
128   reaction(nextstate_start, nextstate_aur, nextstate_idle, nextstate_sdm) {=
129     if nextstate_start.is_present:
130       self.st = nextstate_start.value
131       if self.st == self.states[2]:
132          print("Timeout Scheduled")
133          time_out.schedule(SEC(10))
134          self.timeout_counter += 1timerange
135       self.st = nextstate_aur.value
136     elif nextstate_idle.is_present:
137       self.st = nextstate_idle.value
138     elif nextstate_sdm.is_present:
139       self.st = nextstate_sdm.value
140   =}
141
142   reaction(ia16520) -> ia16520_idle {=
143     if self.st == self.states[0]:
144       ia16520_idle.set(ia16520.value)
145   =}
146
147   reaction(ia16530) -> ia16530_aur {=
148     if self.st == self.states[2]:
149       ia16530_aur.set(ia16520.value)
150   =}
151 }
152
153 reactor Simu {
154   preamble {=
155     import telegrams
156     import db
157     tele = telegrams.Telegrams()
```

```
158   states = ["Idle","Start", "AwaitUserReaction", "SendDocuMessages"]
159 =}
160 input ia16520
161 input ia16530
162 output ia16020
163 output ia16030
164 output timeout
165
166 idle = new Idle()
167 start = new Start()
168 await_user_reaction = new AwaitUserReaction()
169 send_docu_messages = new SendDocuMessages()
170 dis_in = new dispatcher_in()
171
172 ia16520 -> dis_in.ia16520
173 ia16530 -> dis_in.ia16530
174 ia16520 -> send_docu_messages.ia16520
175
176 dis_in.ia16520_idle -> idle.ia16520
177 dis_in.ia16530_aur -> await_user_reaction.ia16530
178
179 idle.timerange -> start.timerange
180 await_user_reaction.send_docu_messages -> send_docu_messages.send_docu_messages
181
182 dis_in.timeout -> timeout
183 send_docu_messages.ia16030 -> ia16030
184
185 idle.nextstate_idle -> dis_in.nextstate_idle after 0
186 start.nextstate_start -> dis_in.nextstate_start after 0
187 await_user_reaction.nextstate_aur -> dis_in.nextstate_aur
188 send_docu_messages.nextstate_sdm -> dis_in.nextstate_sdm after 0
189
190 reaction(start.ia16020, await_user_reaction.ia16020, dis_in.ia16020_timeout) ->
      ia16020 {=
191   if start.ia16020.is_present:
192     ia16020.set(start.ia16020.value)
193   elif await_user_reaction.ia16020.is_present:
194     ia16020.set(await_user_reaction.ia16020.value)
195   elif dis_in.ia16020_timeout.is_present:
196     ia16020.set(dis_in.ia16020_timeout.value)
197 =}
```

```
198 }
```

Listing B.3. Source code Reactors & State Variables Simu

Bibliography

[BLW+22] Soroush Bateni, Marten Lohstroh, Hou Seng Wong, Rohan Tabish, Hokeun Kim, Shaokai Lin, Christian Menard, Cong Liu, and Edward A. Lee. *Xronos: predictable coordination for safety-critical distributed embedded systems*. 2022. arXiv: 2207.09555 [cs.DC]. URL: https://doi.org/10.48550/arXiv.2207.09555.

[Bre00] Eric A Brewer. "Towards robust distributed systems". In: *PODC*. Vol. 7. 10.1145. Portland, OR. 2000, pp. 343477–343502.

[Bre12] 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.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "Sccharts: sequentially constructive statecharts for safety-critical applications: hw/sw-synthesis for a conservative extension of synchronous statecharts". In: *SIGPLAN Not.* 49.6 (June 2014), pp. 372–383. ISSN: 0362-1340. DOI: 10.1145/2666356.2594310. URL: https://doi.org/10.1145/2666356.2594310.

[Hew77] Carl Hewitt. "Viewing control structures as patterns of passing messages". In: *Artificial Intelligence* 8.3 (1977), pp. 323–364. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(77)90033-9. URL: https://www.sciencedirect.com/science/article/pii/0004370277900339.

[LAB+23] Edward A. Lee, Ravi Akella, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. "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.

[Lam19] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 179–196. ISBN: 9781450372701. URL: https://doi.org/10.1145/3335772.3335934.

[LBL+21] Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. *Quantifying and generalizing the cap theorem*. 2021. arXiv: 2109.07771 [cs.DC].

[LBL+23a] Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. "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.

[LBL+23b] Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. "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.

[LL19] Marten Lohstroh and Edward A. Lee. "Deterministic actors". In: *2019 Forum for Specification and Design Languages (FDL)*. 2019, pp. 1–8. DOI: 10.1109/FDL.2019.8876922.

[LMB+21] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. "Toward a lingua franca for deterministic concurrent systems". In: *ACM Trans. Embed. Comput. Syst.* 20.4 (May 2021). ISSN: 1539-9087. DOI: 10.1145/3448128. URL: https://doi.org/10.1145/3448128.

[LMS+20] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A. Lee. "A language for deterministic coordination across multiple timelines". In: *2020 Forum for Specification and Design Languages (FDL)*. 2020, pp. 1–8. DOI: 10.1109/FDL50818.2020.9232939.

[Loh20] Marten Lohstroh. "Reactors: a deterministic model of concurrent computation for reactive systems". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2020. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html.

[LRG+20] Marten Lohstroh, Íñigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. "Reactors: a deterministic model for composable reactive systems". In: *Cyber Physical Systems. Model-Based Design*. Ed. by Roger Chamberlain, Martin Edin Grimheden, and Walid Taha. Cham: Springer International Publishing, 2020, pp. 59–85. ISBN: 978-3-030-41131-2.

[MLB+23] Christian Menard et al. "High-performance deterministic concurrency using lingua franca". In: *CoRR* abs/2301.02444 (2023). DOI: 10.48550/arXiv.2301.02444. arXiv: 2301.02444. URL: https://doi.org/10.48550/arXiv.2301.02444.

[SHL+23] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Marten Lohstroh, Soroush Bateni, and Edward A. Lee. "Modal reactors". In: *CoRR* abs/2301.09597 (2023). DOI: 10.48550/arXiv.2301.09597. arXiv: 2301.09597. URL: https://doi.org/10.48550/arXiv.2301.09597.

[ZLL07] Yang Zhao, Jie Liu, and Edward A. Lee. "A programming model for time-synchronized distributed real-time systems". In: *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. 2007, pp. 259–268. DOI: 10.1109/RTAS.2007.5.