Orchestration of Actor-based Languages for Cyber-physical Systems

Humberto Rodríguez Avila

Dissertation submitted in fulfillment of the
requirement for the degree of Doctor of Sciences

August 2021

Promotors:
Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel
Prof. Dr. Joeri De Koster,Vrije Universiteit Brussel

Jury:
Prof. Dr. Ann Nowé, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Kris Steenhaut, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. Theo D'Hondt, Vrije Universiteit Brussel, Belgium
Prof. Dr. Shigeru Chiba, The University of Tokyo, Japan
Dr. Tim Felgentreff, Oracle Labs, Germany

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 9789493079960
NUR 989

Abstract

Actor-based programming languages already offer many essential features for developing modern cyber-physical systems. These systems exploit the actor model's isolation property to fulfill their performance and scalability demands. Unfortunately, the reliance of the model on isolation as its most fundamental property requires programmers to express complex interaction patterns between actors as complex combinations of asynchronous messages. In the last three decades, several language design proposals have been introduced to reduce the complexity that emerges from describing said interaction and actors' coordination. We argue that none of these proposals is satisfactory to express the many complex interaction patterns between actors found in modern cyber-physical systems.

This dissertation formulates seven smart home automation software construction scenarios (in which every smart home appliance is represented by its own actor) which motivate the need for advanced types of message synchronization patterns between actors in practice; patterns that are lacking in modern distributed actor-based languages. We have collected evidence for the practical relevance of these scenarios by means of an online poll conducted in various online home automation communities. The results of this poll clearly cement the need for advanced synchronization mechanisms in modern actor systems.

A careful analysis of these seven scenarios at the programming language level uncovers five fundamental categories of synchronization patterns. These include 1) the filtering of messages, both based on their content as well as on their timestamps. 2) The selection of one or more messages based on the order in which they arrive. 3) Correlation of messages using logical operators. 4) Accumulation of messages based on windows in time as well as the number of messages. And lastly, 5) Aggregation of accumulated messages.

In this thesis, we present Sparrow, a domain-specific-language (DSL) built on top of Elixir. Sparrow extends the single-message matching paradigm of contemporary actor-based languages to support multiple-message matching. This is enabled by supporting the abstraction and composition of elementary message patterns. Sparrow includes novel language abstractions to support all five categories of synchronization patterns. We also implemented an executable formal calculus for Sparrow – called NEST – that serves as a precise specification of its defining language features. We evaluate our DSL using a quantitative comparison of a state-of-the-art implementation of all seven scenarios with an implementation in Sparrow. Our preliminary evaluation shows that Sparrow effectively reduces the amount of extraneous code that is interleaved with the synchronization code.

Samenvatting

Actorgebaseerde programmeertalen bieden reeds verschillende essentiële constructies aan voor het ontwikkelen van cyber-physical systemen. Zulke systemen benutten het isolatieprincipe van actoren ten volle teneinde hun performantie- en schaalbaarheidseisen te realiseren. Jammer genoeg betekent de afhankelijkheid van 'isolatie als meest fundamentele bouwsteen' van het actormodel meteen ook dat programmeurs gedoemd zijn om complexe interactiepatronen tussen actoren uit te drukken m.b.v. ingewikkelde combinaties van asynchroon gestuurde boodschappen tussen de actoren. In de voorbije drie decennia werden dan ook verschillende taalontwerpen voorgesteld die als doel hebben de complexiteit te reduceren die voortvloeit uit het beschrijven van de coördinatie tussen de actoren met zulke interacties. Wij verdedigen de stelling dat geen van deze voorstellen voldoet om het ingewikkeld soort interactiepatronen uit te drukken die men typisch terugvindt in moderne cyber-physical systemen.

Dit proefschrift formuleert zeven 'smart home automation' softwareconstructiescenario's (waarin ieder 'smart home apparaatje' voorgesteld wordt door zijn eigen actor) die duidelijk de noodzaak aantonen voor meer geavanceerde synchronisatiepatronen van boodschappen uitgewisseld tussen actoren; patronen die dus afwezig zijn in moderne gedistribueerde actorgebaseerde programmeertalen. We hebben bewijs verzameld voor de relevantie van deze scenario's in de praktijk d.v.m. een bevraging in verschillende online 'home automation' gemeenschappen op het internet. Het resultaat van deze bevraging toont overduidelijk de nood aan van meer geavanceerde synchronisatiemechanismen in moderne actorsystemen.

Een gedegen analyse van deze zeven scenario's op programmeertaalniveau heeft niet minder dan vijf fundamentele categorieën van synchronisatiepatronen bloot gelegd. Deze omvatten 1) filteren van boodschappen gebaseerd op hun inhoud en op hun tijdseigenschappen. 2) selecteren

van één of meerdere boodschappen gebaseerd op de volgorde waarin deze werden ontvangen. 3) correlaties van boodschappen m.b.v. logische operatoren. 4) accumulaties van boodschappen gebaseerd op tijdvensters en/of tellingen. 5) aggregaties van geaccumuleerde boodschappen.

In dit proefschrift presenteren we Sparrow, een domeinspecifieke programmeertaal (DSL) die gebouwd werd bovenop Elixir. Sparrow breidt het enkelvoudige bericht-matchingsysteem van een moderne actorgebaseerde taal (in casu dus Elixir) uit met ondersteuning voor meervoudige bericht-matching. Dit wordt verwezenlijkt door het abstraheren en samenstellen van elementaire bericht-patronen. Sparrow biedt taalondersteuning aan voor alle vijf de categoriëen van synchronisatiepatronen die hierboven werden beschreven. Sparrow werd voorzien van een precieze specificatie m.b.v. een formele uitvoerbare calculus, genaamd NEST. We evalueren Sparrow met een kwantitatieve vergelijking tussen een gangbare implementatie van de zeven hogervermelde scenario's met een implementatie in Sparrow. Onze vergelijking toont aan dat Sparrow tot een substantiële reductie leidt van overtollige code die in de gangbare implementaties verstrengeld is met de feitelijke synchronisatiecode.

Acknowledgements

This dissertation would not have been possible without the tremendous support from my promotors, colleagues, friends, and especially my family. I would like to thank my promotors Wolf and Joeri for their guidance all these years. Wolf, thanks for replying to my first email, that email changed the course of my academic journey. I also thank Elisa for her support and guidance at the beginning of this PhD.

Moreover, I would like to thanks all current and ex-SOFTies, especially my REBLs colleagues for all the feedback given me at every research presentation about my work. A big thanks to Thierry for his help to improve the text of this dissertation.

I thank the secretaries of our department who were always ready to help me not only with academic matters but also with my residency related matters every year. Lara, thanks for all your calls and emails to ibz.

I want to sincerely thanks the members of my jury for the time they spent reading this dissertation and their suggestions to improve its final version.

Many thanks to all my friends and teachers who contributed to my education. They are too many to name, but I am thankful for the knowledge and human values that they taught me.

Finally, I would like to thank every member of my big family, and especially my parents for all their support.

Een dikke merci allemaal!
– Humberto

Contents

List of Figures

List of Tables

Listings

List of Acronyms

- **DSL** - Domain-Specific Language
- **CPS** - Cyber-physical systems
- **CEP** - Complex Event Processing
- **LoC** - Lines of Code
- **NEST** - NEST Epitomises Sparrow Theory
- **Hass** - Home Assistant
- **CME** - Communication Model Extensions
- **MV** - Monitor & Verification
- **LS** - Local Synchronization
- **BEAM** - Virtual machine that executes user code in the Erlang Runtime System
- **FIFO** - First In First Out
- **JuPITer** - Join PaTterns Engine
- **AST** - Abstract Syntax Tree
- **IDE** - Integrated Development Environment
- **IQR** - Interquartile Range

CHAPTER 1

Introduction

Cyber-physical systems (CPS) enable technology for numerous innovative applications in fields such as factory automation [55], networked mobility [6], health [88], and smart buildings [51]. These kinds of systems promote a link between physical processes/sensors and a virtual world to facilitate greater productivity, comfort, safety, energy efficiency, and so forth. At the same time, their heterogeneous nature and networked structures impose three challenges for the software industry [4].

- First, these systems often require a more efficient execution by exploiting multi-core processor architectures.
- Second, CPS becomes ever more distributed and event-driven (reactive).
- Third, CPS require a precise *coordination logic*. The coordination logic represents the synchronization abstractions that define and evaluate complex interactions between devices in a CPS (e.g., sensors). For example, the execution of an action in these systems is typically conditioned by the current state or other data shared by more than one device.

These development challenges faced by the software industry have required adopting alternative concurrency models to the widely used

thread-based model. The actor model [35] has become an interesting option to design CPS because it addresses the first two challenges mentioned above. However, in its original form, the model lacks basic synchronization abstractions as indicated by the third requirement.

The interaction and coordination between actors of a system are modelled by exchanging individual asynchronous messages. Whenever an actor is supposed to execute an action in response to a received *set of messages* (rather than just a single message) with certain characteristics, mainstream actor languages (e.g., Erlang, Scala, Elixir) require developers to encode the defining characteristics of the set of messages manually. This limitation puts an extra workload on developers who have to define the coordination logic and manually keep track of relevant data for the coordination process.

Consider the example illustrated in listing 1.1. In this example, the actor will react only if a particular sequence of messages (`:msg_a` → `:msg_b` → `:msg_c`) is received (lines 4, 6, 8). The execution of the reaction code (line 10) is delayed until the expected messages have been received in the correct order. To verify that, the actor must manually keep track of previous messages (lines 5, 7, 12) and validate the progress of subsequent message arrivals in a hard-coded fashion (line 9).

Looking at the simple example shown in listing 1.1 we already observe that the complexity of its coordination logic (line 9) will grow exponentially with the number of actors to coordinate. At the same time, we can imagine that a bigger system will require complex synchronization abstractions and not just sequencing. For example, consider a CPS that ensures an efficient traffic flow; in a way, we never have to stop at a red light unless there is actual cross traffic. In this scenario, *traffic lights* (intersections) and *cars* need to cooperate to achieve such efficient traffic's flow. Both cars and intersections could be modelled as actors. Cars will track their position and communicate with others cars to cooperatively use shared resources such as intersections. In this last scenario, we do not know how many actors must be coordinated, nor the order of the messages to synchronize. Even though this kind of system will require more complex synchronization requirements such as *timing constraints*, current mainstream actor-based languages do not support such constraints even for the synchronization of individual messages.

Briefly, the single-message match mechanism in traditional actors complicates the construction of said CPS. Developers are forced to manually

Listing 1.1 Detect a sequence of messages in Elixir

```
1  def loop({ts_a, ts_b}) do
2      state =
3        receive do
4          {:msg_a, timestamp} ->
5              {timestamp, ts_b}
6          {:msg_b, timestamp} ->
7              {ts_a, timestamp}
8          {:msg_c, timestamp} ->
9              if ts_b > ts_a do
10                 # reaction code
11             end
12             {0,0} # reset state
13       end # receive-end
14     loop(state)
15 end
```

weave two orthogonal concerns of their actor's interactions and coordinations: *when* to react (i.e., precisely describe the set of messages that is supposed to give rise to a certain behaviour) and *how* to react (i.e., the code that describes the actual method to be fired upon reception of said set).

For almost three decades [24, 7, 25], researchers have been developing new programming language features to improve the expressiveness of actors' interaction and coordination features. One particular technique is based on *join patterns*. Join patterns were invented by Benton et al. [12] as part of the join calculus. They were added to the thread-based concurrency model of C# leading to a language called Polyphonic C#. Line 2 in listing 1.2 exemplifies a join pattern that expresses the coordination between the two methods (`Get` and `Put`) of a `Buffer` class. In this example, the calling thread of the `Get` method will be blocked until the asynchronous `Put` method is invoked. The ampersand (`&`) symbol expresses declaratively that *both* threads need to *rendezvous* before the method's body is executed.

Join patterns were recently popularized by Haller et al. [32]. This paper introduces an extension of Scala featuring join patterns called, ScalaJoins. ScalaJoins can be seen as an attempt to transpose the mainly synchronous incarnation of join patterns in Polyphonic C# to the asynchronous world of

Listing 1.2 A join pattern in Polyphonic C#

```
1 public class Buffer {
2     public string Get() & public async Put(string s) {
3         return s;
4     }
5 }
```

Scala. To sum up, join patterns allow developers to express the interaction and coordination logic of a program elegantly.

Although join patterns improved the actor's coordination process significantly, they still fall short in supporting common synchronization abstractions required by modern CPS (see section 2.3). Luckily, a previous study on complex event processing (CEP) systems [16] has revealed a set of wanted properties and operators for correlating events from multiple streams. After distilling a set of essential synchronization abstractions required by modern CPS, we hypothesize that maybe we could borrow well-established correlation operators from the CEP world to enhance join patterns with them.

This dissertation uses smart home scenarios as a particular CPS application domain to help steer our research. We formulate seven smart home automations to motivate the necessity to encode complex interaction and coordination patterns between actors more easily. These examples represent real-world concerns that are really on the radar of the smart home community, as demonstrated by our online survey (see Section 2.2). Using this smart home scenario, we envision that an actor digitally represents a smart device. From these seven automations, we identify *five types of message synchronization abstractions* that cover operations like filtering, selection, ordering, accumulation and transformation of messages.

This dissertation's primary goal is to provide language constructs that aid developers in the coordination process of a group of heterogeneous actors. We ground our research by implementing Sparrow, a dialect of Elixir[1] that features actors whose complex interaction and coordination patterns can be described in a highly declarative fashion. Sparrow's interaction patterns have been harvested from an extensive literature

[1] Elixir can be regarded as a modern Erlang (e.g., with macros) that runs atop BEAM; i.e., the Erlang virtual machine.

study of the state-of-the-art of join patterns and complex event processing systems (see Chapter 3). Hence, Sparrow can be seen as a general-purpose actor language whose actors have been enriched with join patterns and CEP ideas.

1.1 Research Context

The research contexts related to this dissertation are:

Actor-based systems Our research focuses on concurrent and distributed systems whose primary communication and coordination mechanism is to exchange individual messages. As we discussed above, this dissertation explores synchronization abstractions needed by modern actor-based systems.

Coordination models and languages Due to the concurrent nature of actors, the definition of program patterns that deal with their interaction is relevant in all stages of the development process (e.g., debugging and maintenance). In this dissertation, we target coordination primitives that can be incorporated into an actor-based language (e.g., Elixir).

Programming language design In this dissertation, we emphasize the development of a domain specific language (DSL) to aid the coordination process of a group of heterogeneous actors. Since actors are by nature designed to work in conjunction with others, we argue that they required a richer synchronization mechanism than reacting to individual messages.

1.2 Problem Statement

Message passing is the primary communication and coordination mechanism of actor-based systems. Actors read the messages from their inbox one by one . This default behaviour forces developers to manually correlate multiple messages (see chapter 2). Furthermore, the current matching mechanism only filters messages based on their values without taking into account time constraints. It is up to the developers to enforce such

advanced message synchronization abstraction themselves in a hard-code way.

This dissertation motivates a subset of advanced message synchronization abstractions using several motivating examples drawn from the domain of smart home automation. However, their need can also be observed in other domains of cyber-physical systems [37]. We drew our inspiration for such our subset of message synchronization abstractions from complex event processing systems. This dissertation also explores how to integrate these synchronization abstractions into the message matching mechanism of actors. Listing 1.3 gives a sneak peek of how we envision such integration.

Listing 1.3 Vision of an advanced message synchronization abstraction for actor-based languages

```
1  pattern kw_alert as {:kw, @value}[window: {3, :weeks}]
2                  |> fold(0, fn({_,_,v}, acc)-> acc+v end)
3                  |> bind(total)
4                  |> total > 200
```

In the snippet code of listing 1.3, an actor defines a pattern that will match and accumulate the daily electricity consumption messages for three weeks (line 1). After summing up the readings (line 2), the pattern will determine if the total consumption was greater than 200 kWh (line 4). Although it is not shown in the example, this kind of abstraction will allow developers to define a set of actions that should be executed when its conditions are satisfied.

In summary, this dissertation aims to tackle the following problem:

> Actors often need to correlate messages from different sources **in a particular order**, and **within a specific time window**. The matching mechanisms present in mainstream actor languages only filter messages based on their values.

1.3 Research Goal

This dissertation aims to *extend* the actor model's default message matching mechanism to support the *detection of complex message synchronization ab-*

stractions. Particularly, we aim to provide a rich set of message correlation operators. To achieve this goal, we explore the integration of well-known CEP operators into the actor model (see chapter 3). We envision the following requirements for our extension of the actor model:

1. It must seamlessly integrate into a canonical actor-based host language.
2. It must provide advanced message filters and selection mechanisms.
3. It must combine multiple operators by which developers can define complex messages correlation.
4. It must improve the code's expressiveness to correlate multiple messages.

1.4 Approach

This dissertation proposes a novel domain-specific language (DSL), called Sparrow, that tackles the problem stated in Section 1.2. We envision its development in five steps:

- First, we **distil common message synchronization abstractions** found "in the wild" in the smart home community (see Section 2.1).
- Second, we use an online poll to **confirm how frequently** the distilled message **synchronization abstractions appeared** in real home automations of the smart home community (see section 2.2).
- Third, we **investigate the support** of the identified message synchronization abstractions by state-of-the-art proposals of both join patterns (see section 3.1) and complex event processing (see section 3.2). This study will allow us to detect the weaknesses/strengths of both approaches and apply this knowledge to implement Sparrow.
- Fourth, we **implement a DSL prototype** on top of a mainstream actor language (see section 4.2).

- Fifth, we also incrementally **formalized the operational semantics** of our DSL using Redex [21] to experiment with each message synchronization abstraction (see section 5.1).
- Sixth, we **compare the solutions** obtained with our DSL against the ones of two mainstream smart home platforms and an actor language (see section 7.2).

1.5 Contributions

This dissertation makes the following contributions:

A suite of common synchronization requirements. A proposal for a suite of common synchronization requirements needed by modern actor systems. The need for these requirements was confirmed by more than 700 developers from the smart home community.

A survey of existing actor-based coordination approaches and CEP operators. An extensive survey of related work on coordination abstractions for the actor model and complex-event processing (CEP) operators. From this survey, we conclude that traditional implementations and extensions to the actor model have limited support for the synchronization requirements identified in this dissertation. However, these requirements were commonly tackled by CEP operators.

A domain-specific language for advanced coordination of heterogeneous actors. Sparrow, a novel domain-specific language as a technical incarnation of the aforementioned synchronization requirements on top of Elixir. To the best of our knowledge, Sparrow is the first actor-based language to combine join patterns and complex event processing techniques.

A formal calculus of Sparrow called NEST and its mechanization. A specification of the operational semantics of Sparrow, called NEST. This formal specification is used to experiment with its synchronization abstractions.

A novel RETE-based matching algorithm. A custom implementation of the RETE [22] algorithm to reduce Sparrow's message matching performance overhead by supporting an incremental matching mechanism.

Basic Tools Support Basic software tools to support the development and debugging of Sparrow-based programs.

The implementation of Sparrow, its formal semantics in Redex, and the examples used for its validation are available at `http://soft.vub.ac.be/~hrguez/sparrow-lang`.

1.5.1 Publications

Parts of this dissertation appear in the following publications:

Article

- Advanced Join Patterns for the Actor Model based on CEP Techniques. Humberto Rodriguez Avila, Joeri De Koster, and Wolfgang De Meuter. The Art, Science, and Engineering of Programming, 2021, Vol. 5, Issue 2, Article 10. DOI: `https://doi.org/10.22152/programming-journal.org/2021/5/10`

  This paper describes a subset of CEP-based synchronization operators implemented into an actor-based language to facilitate the coordination between actors.

Workshop Paper

- Sparrow: a DSL for coordinating large groups of heterogeneous actors. *Humberto Rodriguez Avila*, Joeri De Koster, and Wolfgang De Meuter. In Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2017). Association for Computing Machinery, New York, NY, USA, 31–40. DOI: `https://doi.org/10.1145/3141834.3141838`

  This workshop paper introduces our first attempt at a novel coordination model for actor-based programs. This first prototype illustrates the bases of Sparrow's message correlation primitives.

Poster & Demo & Talk

- An Elixir library for programming concurrent and distributed embedded systems. *Humberto Rodriguez Avila*, Elisa Gonzalez Boix, and Wolfgang De Meuter. In Companion to the First International Conference on the Art, Science and Engineering of Programming

(Programming 2017). Association for Computing Machinery, New York, NY, USA, Article 6, 1. DOI: `https://doi.org/10.1145/3079368.3079383`

This demo introduces a small actor library for developing concurrent and distributed embedded systems. We showcase the implementation details of a classic leader election algorithm using a small cluster of Raspberry Pi computers.

- Intentional Join Patterns for Coordinating Large Groups of Heterogenous Actors. *Humberto Rodriguez Avila*, Elisa Gonzalez Boix, and Wolfgang De Meuter. Poster session presented at the International Conference on the Art, Science and Engineering of Programming (Programming 2017).

  This poster introduces our initial steps towards the Sparrow DSL. We present a limited join pattern DSL to correlate multiple messages.

- Tierless Reactive Programming for Big Data: Tackling the Storage-Signal Impedance Mismatch. *Humberto Rodriguez Avila*, Elisa Gonzalez Boix, and Wolfgang De Meuter. Poster session presented at European Conference on Object-Oriented Programming (ECOOP 2016)

  This poster illustrates a reactive approach for processing a large amount of data using NO-SQL databases. We discuss an extension to Riak DB and a SQL-like DSL to react to incoming data.

- Reactive Databases for BigData Applications. *Humberto Rodriguez Avila*, Wolfgang De Meuter, Elisa Gonzalez Boix. Talk at the 4th Graph-based Technologies and Applications Workshop (GRAPH-TA 2016).

  In this talk, we introduce an extension to Riak DB and a SQL-like DSL to react to incoming data in NO-SQL databases.

Additionally, I contributed to the following publication during my research:

- Composable higher-order reactors as the basis for a live reactive programming environment. Bjarno Oeyen, *Humberto Rodriguez Avila*, Sam Van den Vonder, and Wolfgang De Meuter. In Proceedings of

the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2018). Association for Computing Machinery, New York, NY, USA, 51–60. DOI: `https://doi.org/10.1145/3281278.3281284`

This paper introduces a new reactive language based on first-class higher-order reactors, called Haai.

1.6 Roadmap

This dissertation is organized as follows:

Chapter 2: Motivation uses a smart home scenario to distil five categories of synchronization requirements that are needed to express common interaction and coordination patterns between actors.

Chapter 3: Coordination of Actors and CEP Operators analyzes the support of the synchronization requirements described in Chapter 2 by actor-based technologies and complex event processing (CEP) operators.

Chapter 4: Sparrow: A DSL for Actor Coordination introduces the main synchronization abstractions proposed by our DSL to support the synchronizations requirements described in Chapter 2.

Chapter 5: NEST: A Formal Semantics of Sparrow describes a formal calculus for Sparrow, called NEST. This formalism serves as a precise definition of the semantics of core Sparrow abstractions.

Chapter 6: Sparrow: An Elixir DSL Implementation reveals implementation details behind the synchronization operators supported by Sparrow. We explain the variant of the RETE [22] algorithm used to provide an incremental matching of Sparrow's patterns. Furthermore, this chapter describes an extension to the Visual Studio Code editor to support incremental static analysis, code autocomplete, and inline report of build warnings and errors of Sparrow-based programs. This chapter also introduces a real-time inspector that can be used to debug Sparrow-based applications.

Chapter 7: Validation details a quantitative evaluation based on the implementation of the seven smart home scenarios described in

Section 2.1. Particularly, it compares our DSL solutions against the ones of two smart home platforms (i.e., openHAB and Hass) and a modern actor language (i.e., Elixir).

Chapter 8: Conclusions concludes this dissertation and highlights avenues for future work.

CHAPTER 2

Motivation

Cyber-physical systems and particularly smart home systems exhibit a wide range of synchronization requirements. Devices used in such systems exhibit three main properties that fulfil our vision of modelling them as actors. First, they are heterogeneous by nature (e.g., bulbs, thermostats, motion sensors). Second, they are distributed and isolated entities that communicate with each other directly or through a middleman (i.e., local or cloud server). Third, their state and behaviour are modified based on coordinated actions within a group.

We adopt a smart home use case to distil common synchronization requirements found on daily automation rules. Section 2.1 introduces seven smart home scenarios that embody different coordination problems between a group of actors representing smart devices. Later, section 2.2 reports on an online survey across multiple smart home communities which we used to validate the coordination problems exemplified by our scenarios. Finally, section 2.3 describes five categories of synchronization requirements needed to express those coordination problems.

2.1 Smart Home Scenarios

Smart homes are an application domain where complex interactions between different smart devices occur. Home automation rules correlate

events coming from multiple devices in order to execute a particular action (e.g., turn on the lights). Here, we present seven home automation scenarios that exemplify basic form of synchronization between a group of heterogeneous actors. These examples were inspired by real automation rules shared on community forums of smart home platforms (e.g., openHAB[1] and Hass[2]):

1. Turn on the lights in a room if someone enters, and the ambient light is less than 40 lux.
2. Turn off the lights in a room after two minutes without detecting any movement.
3. Send a notification when a window has been open for over an hour.
4. Send a notification if someone presses the doorbell, but only if no notification was already sent in the past 30 seconds.[3]
5. Detect home arrival or leaving based on a particular sequence of messages, and activate the corresponding "scene"[4]: *occupied-home* or *empty-home*.
6. Send a notification if the combined electricity consumption of the past three weeks is greater than 200 kWh.
7. Send a notification if the boiler fires three "Floor Heating Failures" and one "Internal Failures" within the past hour, but only if no notification was sent in the past hour.

Listing 2.1 shows an implementation of scenario 5 to illustrate the degree of complexity that users of these platforms have to handle by themselves. This openHAB implementation considers two motion sensors, one in the entrance hall (α), and the second one outside the front door (δ). Furthermore, we use a contact sensor (β) to detect when the front door was opened. The *occupied-home* scene is enabled by the following sequence of messages $\delta \rightarrow \beta \rightarrow \alpha$, and the *empty-home* scene by $\alpha \rightarrow \beta \rightarrow \delta$. Both scenes will be activated if the three messages occur in a time window of 60

[1] openHAB Forum (`https://community.openhab.org`, last accessed 2020-10-01).
[2] Hass Forum (`https://community.home-assistant.io`, last accessed 2020-10-01).
[3] The postman always rings twice.
[4] Scenes are used for setting a group of values or devices' states.

Listing 2.1 Jython-script implementation for scenario 5 in openHAB

```
c_door = ZDT.now().minusHours(24)
m_hall = ZDT.now().minusHours(24)
m_door = ZDT.now().minusHours(24)

@rule("(Py) Front Door Opened")
@when("Item Front_Door_Contact changed to OPEN")
def front_door_opened(e):
    global c_door
    c_door = ZDT.now()

@rule("(Py) Motion Detected - Entrance Hall")
@when("Item Entrance_Hall_Motion changed to ON")
def entrance_hall_motion(e):
    global m_hall, m_door, c_door
    m_hall = ZDT.now()

    if m_door.isBefore(m_hall.minusSeconds(60)):
        return

    if m_hall.isAfter(c_door) and c_door.isAfter(m_door):
        # code logic for arriving home

@rule("(Py) Motion Detected - Front Door")
@when("Item Front_Door_Motion changed to ON")
def front_door_motion(e):
    global m_hall, m_door, c_door
    m_door = ZDT.now()

    if m_hall.isBefore(m_door.minusSeconds(60)):
        return

    if m_door.isAfter(c_door) and c_door.isAfter(m_hall):
        # code logic for leaving Home
```

seconds. Although this time constraint is not mentioned in the description of scenario 5, we added it to our implementation to discard old messages. Similarly, this particular openHAB implementation also considers that the home has only one inhabitant. Although our solution was improved thanks to the feedback received from the openHAB community [61], *it exposes a*

lack of programming abstractions to synchronize messages from multiple devices.

- First, we were responsible for keeping track of each sensor's last update (β line 9, α line 15, δ line 27).
- Second, we manually discarded messages older than the 60 seconds time window (lines 17, 29).
- Finally, we had to verify that the messages were received in the right order (lines 20, 32) before executing the automation's reaction.

In summary, scenario 5 can be implemented in 33 lines of code (LoC) in openHAB. Of those 33 LoC, 24 are part of the setup and coordination logic of the different devices involved. This dissertation hypothesizes that a set of novel synchronization primitives can reduce the required LoC to implement the above synchronization concerns and improve the code expressiveness of their solutions.

2.2 Proof of Scenarios' Relevance

To validate that each of our scenarios is a representative example of the different synchronization requirements found in the wild, we ran an online poll. We asked each respondent to reply with a yes-or-no answer whether or not they even felt the need for automations similar to each of our seven scenarios. Table 2.1 shows the questionnaire of our online poll. As we can notice, each question of our poll represents one of our scenarios. In the rest of this section, we will refer to them as questions or scenarios interchangeably.

Our poll was published in four smart home community forums (Hass [58], openHAB [60], SmartThings [62], and Hubitat [59]). In one month, it accumulated votes of 714 contributors. Figure 2.1 shows the results of this poll. Even though we cannot derive any statistical meaningful conclusions based on a mere poll, they suggest that our scenarios are good examples of concerns that live within the smart home community. On the one hand, scenario 1 (Q1) exemplifies the need for *synchronizing* multiple *independent* messages. In this scenario, there are two sensors (motion and light) whose messages need to be synchronized. This concern was identified and raised as a known issue by 662 (92.72 %) of the voters. On the other hand,

Table 2.1 Online poll questionnaire

Question	Description
Q1	I have automations that involve multiple devices and conditions. For example, turn on the lights of a room if motion is detected and its ambient light is less that 40 lux.
Q2	I have automations that require to react to the absence of events. For example, turn off the lights in a room after two minutes without detecting any movement.
Q3	I have automations that require do some action if a device remains in the same state for a period of time. For example, send a notification when a window has been open for over an hour.
Q4	I have automations that require ignore some repeated events within a period of time. For example, send a notification if someone presses the doorbell, but only if no notification was already sent in the past 30 seconds.
Q5	I have automations that require to detect a particular sequence of events. For example, detect home arrival or leaving based on a particular sequence of messages, and activate the corresponding scene.
Q6	I have automations that their conditions are based on specific historical data of a device or multiple devices. For example, send a notification if the combined electricity consumption of the past three weeks is greater than 200 kWh.
Q7	I have automations that require a particular number of events before doing some action. For example, send a notification if the boiler fires three Floor Heating Failures and one Internal Failure within the past hour, but only if no notification was sent in the past hour.

advanced time-based synchronization of messages like the ones of scenarios 2 and 3 was also a well-recognized concern by more than 515 (>72 %) respondents. Scenarios 4, and 5 also rely on fairly advanced time-based synchronization with additional constraints: *discard* messages and enforce a particular *order* in which its constituent messages must match. In both cases, more than 295 (>40 %) of the voters was familiar with the need for such constraints. Finally, even though scenarios 6 and 7 target expensive devices which currently offer limited integration with third-party platforms, more than a quarter of the respondents identified the need for *aggregating* multiple messages over a specific *time window*.

The results of our poll do not validate our list of scenarios as exhaustive. However, they do give us the certainty that the listed scenarios represent common synchronization requirements demanded by automation rules found "in the wild". We implemented our scenarios in two *popular* open-source smart home platforms (i.e., openHAB and Hass), and a *modern* actor based language (i.e., Elixir). The source code of these implementations can be found online in our GitHub repository [56]. Despite the clear need for such requirements, we could not get a straightforward and elegant implementation for them in our solutions. Even though the selected programming environments offer at least the *same* synchronization abstrac-

	Smart Home Communities				
	openHAB	Hass	SmartThings	Hubitat	Total Votes
Q1	440	134	55	33	662
Q2	333	109	51	27	520
Q3	332	107	50	23	512
Q4	228	66	31	17	342
Q5	210	47	25	16	298
Q6	169	39	20	11	239
Q7	132	31	17	11	191

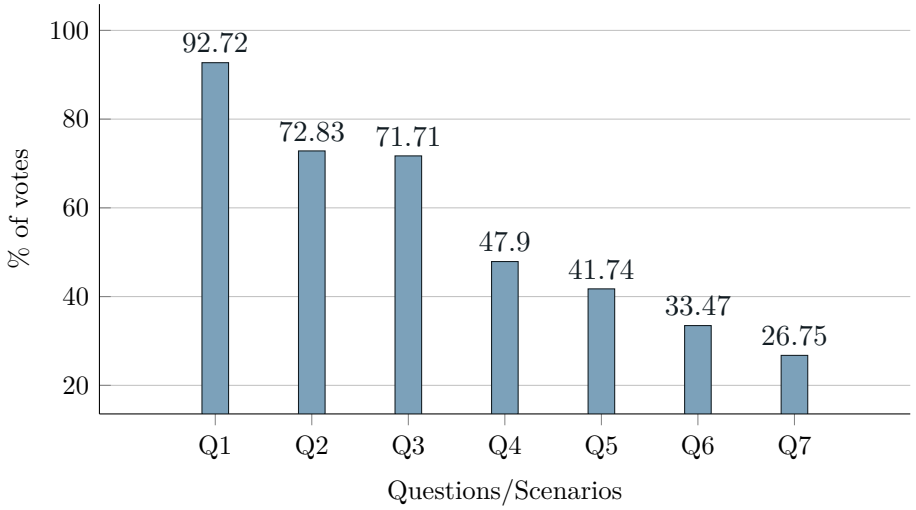


Figure 2.1 Online poll results. Voters: 714. Voting time: 1 month

tions as their state-of-the-art alternatives (e.g., SmartThings, Akka/Scala), they lack advanced synchronization abstractions to coordinate a group of heterogeneous devices.

2.3 Message Synchronization Requirements

In this dissertation, we envision an extra layer of *synchronization logic* for the actor's inbox. This layer should enhance actors' current message matching mechanism to facilitate the interaction and coordination of a group of actors. Further analysis of our scenarios allowed us to disambiguate five categories of synchronization requirements needed to express the scenarios. These requirements are:

Support for filtering is needed to enable the filtering of messages based on their attribute values or timing constraints. For example, in scenario 1, we are required to filter out messages from the ambient light sensor for which the value does not reach 40 lux. Additionally, scenario 4 and 6 show the need for filter capabilities based on the absence or presence of messages within a certain time window respectively.

Support for selection is needed to choose messages from the list of matching messages after filtering. Traditional actor languages only allow for the consumption of the oldest (first in) message. However, in some cases, we require more *flexible* message selection policies. For example, in scenario 7, we are only interested in the latest *Internal Failure* message, and the third *Floor Heating Failure* message. In general, expressing these types of constraints will require selection abstractions that allow us to select any message from the list of matching messages after filtering.

Support for correlation is needed to match a list of different types of messages and to unify their attributes. For example, in scenario 1, we are required to match a motion sensor message and an ambient light sensor message but only match if both are present. Additionally, in scenario 5, we also have to specify the order in which those messages arrived to trigger the right scene. Depending on the order in which sensors detected movement, the person either left the house or just arrived home.

Support for accumulation is needed to aggregate a list of matching messages (e.g., after filtering) for further processing. For example, in scenario 6, we have to aggregate electricity consumption messages of the last three weeks into their sum.

Support for transformation is needed to transform (e.g., map) such a list of accumulated messages. These messages can be subject to new predicate conditions. For example, in scenario 6, we need to check if the total electricity consumption is greater than 200 kWh before sending a notification. Only if the predicate condition is true all the accumulated messages are consumed.

2.4 Conclusion

The synchronization of messages plays a significant role for cyber-physical systems. Using seven smart home scenarios, we identified five primary categories of synchronization requirements needed to coordinate a group of smart home devices. Although we have demonstrated the need for such synchronizations requirements, other application domains might need different synchronization requirements.

We used two mainstream smart home platforms and a modern actor-based language to implement our scenarios. Even though our technologies choices were state-of-the-art in their domain, we observed a lack of synchronization primitives to coordinate a group of smart devices.

CHAPTER 3

Coordination of Actors and CEP Operators

This chapter analyzes state-of-the-art technologies and their support for the synchronization requirements described in chapter 2. Particularly, we focus on actor-based technologies and complex event processing (CEP) systems. In section 3.1, we analyze the different synchronization abstractions supported by actor-based languages/frameworks. Section 3.2 instead explores synchronization abstractions implemented by general-purpose CEP technologies.

3.1 Coordination of Actor-based Systems

In this section, we analyze the different synchronization abstractions supported by implementations of the original actor model [35] and its extensions. Furthermore, we discuss the current support for the kind of synchronization requirements described in section 2.3 by actor-based technologies. First, we describe built-in synchronization abstractions of the canonical actor model (see section 3.1.1). Second, we analyze a set of extensions to the original actor model. Based on the type of of extension they propose, we group such solutions into three main categories: *communication model extension* (see section 3.1.2), *monitor & verification* (see section 3.1.3), and *local synchronization* (see section 3.1.4). Since our

definitions of categories are not mutually exclusive, some proposals may fit in more than one category.

3.1.1 The canonical Actor Model

Since its proposal in 1973 [35], implementations of the actor model have used *message exchanging* as the coordination mechanism for the constituents of a system. Historic actor implementations such as PLASMA [34], ABCL/1 [87], and Rossete [79] introduced basic concepts that have been adopted by modern actor languages (e.g., Elixir [76]). Here we list them:

- First, *messages* are the unit of communication between a group of actors.
- Second, the *inbox* of an actor stores the messages in the order received by that actor.
- Third, the *interface* of an actor defines the set of *message patterns* that the actor understands. Only messages that satisfy the actor's interface will be processed.
- Fourth, a *message pattern* defines a particular type of message and its set of attributes to match. For example, the Elixir's message pattern `{:motion, id, status, location}`, matches messages of type `:motion` that consist of three attributes `id`, `status`, and `location`. Additionally, messages patterns can filter specific attributes using predicate expressions (a.k.a. guards) or pattern-matching techniques. In functional languages such as PLASMA, Erlang [73], Scala/Akka [75], and Elixir such messages patterns are commonly implemented as *clauses* of a matching primitive (e.g., Elixir's *receive*). In contrast, object-based implementations such as ABCL/1, Rossete, AmbientTalk [82], and Pony [13] declare them as methods.
- Fifth, messages that satisfy a particular message pattern are removed from the actor's inbox and *consumed* by the body (a.k.a. *reaction*) of the message pattern.
- Six, the *body* or *reaction* of a message pattern defines the set of actions to be executed after a message match is found. Both messages patterns and reactions are *coupled* during the definition of the former.

Table 3.1 Synchronization requirements supported by canonical actor implementations

	Filter		**Selection**	**Correlation**			**Accum.**		**Transf.**
	Content-based	Time-based	Flexible	Conjunction	Disjunction	Sequencing	Count-based	Time-based	Aggregation
PLASMA [34]	x	-	-	-	-	-	-	-	-
ABCL/1 [87]	x	-	-	-	-	-	-	-	-
Rossete [79]	x	-	-	-	-	-	-	-	-
Erlang [73]	x	-	-	-	-	-	-	-	-
Elixir [76]	x	-	-	-	-	-	-	-	-
Scala/Akka [75]	x	-	-	x[a]	-	-	-	-	-
AmbientTalk [82]	x	-	-	x[a]	-	-	-	-	-
Pony [13]	x	-	-	-	-	-	-	-	-

[a] Conjunction is only enforced to the synchronization of a group of futures

Table 3.1 summarizes the support of the synchronization requirements identified in section 2.3 by the actor-based languages analyzed in this section. At first sight, observe that all languages support filter abstractions over the content (*attributes*) of a message (see column *Filter* in table 3.1). These languages mostly based these abstractions on *pattern-matching* techniques. However, implementations such as Erlang, Elixir, and AmbientTalk support guard expressions to complement pattern-matching and define complex filter expressions. For example, listing 3.1 shows the definition of two messages patterns in Elixir. The first one illustrates pattern-matching filtering. In this case, the message pattern defined in line 2 will match `:motion` messages if their last attribute (`location`) is equal to the atom[1] `:bedroom`. The second message pattern shows a more complex message filter based on pattern-matching and a guard expression (line 4). This message pattern will match `:temperature` messages if their last attribute (`location`) is equal to the atom `:bedroom`, and the attribute `value` is less than 18 ℃. However, none of these languages supports abstractions to filter messages based on *time constraints*. They also completely lack support for a *flexible selection* of messages (see column *Selection*). In these languages, message patterns always try to match the oldest message in the actor's inbox. Furthermore, their message patterns only match individual mes-

[1] Atoms are constants whose values are their own name.

Listing 3.1 Message pattern examples in Elixir

```
1  # message patterns
2  {:motion, id, status, :bedroom} ->
3      # reaction code
4  {:temperature, id, value, :bedroom} when value < 18 ->
5      # reaction code
```

sages. Therefore accumulation (see column *Accum.*) and transformation (see column *Transf.*) of messages are not supported. At the same time, only Scala/Akka and AmbientTalk support a *rudimentary* form of message correlation based on abstractions to wait for the resolution of a group of futures (see column *Correlation/Conjunction*).

3.1.2 Communication Model Extensions

Proposals in this category focus mostly on two types of modifications to the traditional actor's communication model. On the one hand, we have proposals that add an extra synchronization layer to the traditional actor's point-to-point communication model [84, 40]. On the other hand, we have proposals that add support for multi-cast message communication between actors [7, 30, 67, 27, 17, 29]. This second type of extension enriches actors systems with two main features: *anonymity* and *open-endedness* [17]. The former allows actors to dynamically select potential receivers of a single message, by relying on predicates over their exposed attributes. The latter allows actors to enter or leave a system at any time. In the rest of this section, we briefly describe each of these proposals and summarize their support for the synchronization requirements identified in section 2.3.

Directors [84] offer a hierarchical coordination model where a *director* constrains the interaction of a group of actors, called a *cast*. A director is a particular type of actor that guarantees the reception of a message by a potential receiver only if it satisfies its constraints. This hierarchical approach requires that the sender explicitly name the target of the message. In this extension, actors must explicitly join a cast, but they can only belong to *one* or *none* at any time. Furthermore, a director can belong to a cast being coordinated by another director. In such hierarchical cases, a message will be delivered to a target actor only after approval by all the directors involved in its coordination forest path.

Coordinators [40] introduce a special type of actor named *coordinators* to handle variabilities in a product line of actor-based systems. Coordinators are like any other regular actors with the exception that their behaviours are defined using Reo [10] circuits to express the coordination logic. This particular property allows coordinators to keep the variability logic of the system separated from the computational components (a.k.a. actors). However, this approach also inherits the static nature of Reo in the sense that coordinators' behaviour can not be changed dynamically. Furthermore, the circuit flow created by a coordinator processes a single message at a time.

ActorSpace [7] proposes a new communication model inspired by the Linda [28] model based on *destination patterns*. This particular type of patterns allows actors to specify a group of potential recipients of a message abstractly. To support that, ActorSpace provides two communication primitives: `send` and `broadcast`. The former sends a message to one of the actors matching a pattern specified by the sender. The selection of the receiver actor is done in a non-deterministic way. The latter primitive sends a message to all actors matching a pattern specified by the sender. Both primitives exploit the visible attributes of an actor, which provide an abstract view of an actor. This proposal also introduces an abstraction to scope the mechanism for pattern-matching, called *actor spaces*. Unlike other Linda-based models, actor spaces provide a more secure communication mechanism since the message's sender limits its potential receivers. Upon the reception of a message, an actor space checks the destination pattern against the list of attributes of its registered actors before forwarding the message.

The Reflective Communication Framework (RCF) [30] allows dynamic customization of *communication protocols* (e.g., message priority) between actors. RCF's actors can change their communication protocol at any time since the model separates its specification from its implementation. However, like in the traditional actor model a protocol process a single message at a time. RCF uses a particular type of meta-actors called *messengers* to control the runtime behaviour of application-level actors. Each actor has a corresponding meta-actor or messenger. A messenger serves as a customized and transparent inbox that can also modify its actor's state. Messengers extend the single inbox of the actor model with

unique inboxes to attach communication protocols to both incoming and outgoing messages.

TOTAM [67], like ActorSpace, extended the Linda model to support a dynamic scoping mechanism that limits the transportation of tuples (messages) in mobile ad hoc networks. Similarly to ActorSpace's destination patterns, TOTAM uses *tuple space descriptors* to scope tuples and prevent their propagation to unwanted locations. By using such descriptors TOTAM-based applications are able to enhance privacy and reduce their network traffic.

Syndicate [27] introduces a *publish-subscribe* mechanism to generalize the point-to-point communication of the actor model. A Syndicate's program may consist of two types of actors: a *leaf-actor* or a *network-actor*. The former resembles a traditional actor [35], but they may also publish state change notifications in its *network's shared dataspace*. The latter groups a particular set of communicating actors. Each leaf-actor participates in scoped conversations, which act as a *message bus* for the exchange of messages and coordinate access to a shared dataspace. A leaf-actor uses assertions (*subscriptions*) to express its interest in particular messages. Assertions are stored in the network-actor in which the leaf-actor is a member. A network-actor keeps a list of their current leaf-actors and notifies them each time there is a new match for their assertions. Unlike Linda's tuple spaces, network-actors support *continuous queries* over their shared dataspaces. Furthermore, actor's assertions exist only as long as their leaf-actors live. Once a leaf-actor crashes, all its assertions are automatically retracted from its network-actor.

interActors [29] add an extra communication layer to the traditional actor model to support complex communication protocols (e.g., broadcaster, router) based on a reflective approach. Protocols are implemented as first-class objects called *Communications*. Communications are represented by two or more meta-actors of type *outlet* and *handler*. On the one hand, outlets can receive (*input-outlet*) and send (*output-outlet*) messages. On the other hand, handlers carry out a particular communication logic to process the received messages. Furthermore, they can send messages to other handlers or outlets. A *communication* always requires at least one input and one output outlet. However, it might not need a handler if any interaction logic is required. Actors interact with a *communication* only by sending messages to known input-outlets and receiving messages from

```
A to("language = this.interest") ! {Language, Id}

B from("hobby = this.hobby and language = this.interest"),
  receive
    {Language, Buddy} ->
       to("id = $Buddy") ! {ok, Id}
  end

C Count = to("language = this.interest") ! {Language, Id}
  from("hobby = this.hobby and language = this.interest", Count),
  receive
    {Language, Buddy} ->
       to("id = $Buddy") ! {ok, Id}
  end
```

Figure 3.1 Example of synchronization abstractions in AErlang

output-outlets. Outlets and handlers declare a particular behaviour to define how to react to incoming messages. Behaviours, like the traditional actor's message patterns match and process a single message at a time.

AErlang [17] extends Erlang's actor model with a set of primitives to support attribute-based communication [3]. Like Syndicate, AErlang uses a centralized mechanism (*broker*) to select groups of communicating actors dynamically. AErlang's multicast communication between actors relies on predicates matches over their exposed attributes. Its predicates can be seen as a combination of ActorSpace' *destination patterns* and Syndicate's assertions. Predicates can specify attribute's conditions that must be satisfied by both potential *receivers* or *senders* of a message. However, unlike Syndicate's assertions, these predicates are based on a traditional (*non-continuous*) database query. Whenever a process sends a new message, the AErlang system's broker parses the predicates and converts them into a database query. After that, it forwards the message to all the receivers that satisfied the query. Figure 3.1 illustrates the three main synchronization abstractions supported by AErlang. The examples shown in this figure are copied from [17]. Figure 3.1.A shows a snippet to send a message to all members (actors) of a social network who share the same language interests than the sending actor. Here, AErlang extends the Erlang's send `!` primitive to receive as its first argument a boolean predicate in order to determine potential receivers of the message, instead of just the actor id. Predicate expressions can access the actor's attributes using the `this` operator (e.g., `this.hobby`). In contrast, figure 3.1.B uses the primitive `from` to limit the reception of messages of an actor. Predicate expressions

can also access local variables using the `$` operator (e.g., `$Buddy`). In this second example, the actor will only receive messages from senders who share its hobby and language preferences. Notice that in our previous example, we omitted the definitions of attributes (e.g., `interest`) and local variables (e.g., `Language` and `Id`).

In general, languages/frameworks in this category use a centralized actor to filter messages based on their values or the actor's attributes. However, none of them supports time-based filtering of messages to specify which message to match if more than one is available. Furthermore, these proposals do not provide abstractions to detect the absence of messages. Additionally, a particular matching order cannot be specified neither a message pattern that matches a disjunction of messages. Finally, all of the synchronization mechanisms in this category are only able to match individual messages, except for interActors and AErlang. In interActors, the sender of a message can wait for two or more *reply* messages (a.k.a. futures) before continuing. AErlang instead allows the receiver actor to accumulate and filter a finite number of messages of the *same* type. For example, in figure 3.1.C the variable `Count` stores the total number of receivers of the message `{Language, Id}`. Later, this variable is used by the `from` primitive to accumulate all the responses. Only after all expected responses have arrived, the `receiver` block is executed. Although this `from` variant allows developers to accumulate messages, it can block the actor forever if one of the potential receivers do not reply.

Table 3.2 summarizes the supported synchronization requirements defined in section 2.3 by Communication Model Extensions (CME) proposals.

3.1.3 Monitor & Verification

The proposals described in this section use reflection techniques to monitor and limit the interaction of one or a group of actors [24, 52, 45, 42, 53, 68, 19, 66, 46, 23, 64, 65, 33]. In this category, the coordination process is mostly done by a special type of meta-actor which enforces a particular protocol for incoming and outgoing messages. In the rest of this section, we briefly describe each of these proposals and summarize their support for the synchronization requirements identified in section 2.3.

Synchronizers [24] uses reflection to observe and limit the delivery of messages for a group of actors. Synchronizers can be seen as meta-actors that express *coordination patterns* in form of multi-actor constraints to

Table 3.2 Synchronization requirements addressed by CME proposals

	Filter		**Selection**	**Correlation**			**Accum.**		**Transf.**
	Content-based	Time-based	Flexible	Conjunction	Disjunction	Sequencing	Count-based	Time-based	Aggregation
ActorSpace [7]	x	-	-	-	-	-	-	-	-
TOTAM [67]	x	-	-	-	-	-	-	-	-
Directors [84]	x	-	-	-	-	-	-	-	-
Syndicate [27]	x	-	-	-	-	-	-	-	-
AErlang [17]	x	-	-	x[a]	-	-	-	-	-
interActors [29]	x	-	-	x[a]	-	-	-	-	-
RCF [30]	x	-	-	-	-	-	-	-	-
Coordinators [40]	x	-	-	-	-	-	-	-	x

[a] Conjunction is only enforced to invocation of messages

limit the interaction between them. This particular type of constraints is called *atomicity constraints*. These constraints wait for a set of messages to be available before dispatching them as a whole and *without temporal ordering*. Atomicity constraints can express under which conditions an actor can or can not handle a message. However, they can interfere with each other because multiple synchronizers can constrain the same actor.

RTsyncronizers [52] extend Synchronizers' coordination patterns with quantitative constraints, called *timing relations*. Like its predecessor, RTsyncronizers pursue a total separation between the communication and coordination concerns of an actor system. Furthermore, they facilitate the ability to modify real-time message constraints over a group of actors dynamically. Additionally, an exception can be signaled if the coordination pattern can not be resolved. In this way, messages are not delayed forever. However, unlike Synchronizers, the RTsynchronizers' computation model is based on the assumption that actors of a system have their local clocks synchronized, and their invocations are scheduled atomically.

Moses [45] proposes the concept of regulated coordination policies for actors. Policies are defined as declarative coordination constraints and they are enforced on all actors of a system. Each policy is defined by a set of rules called, *Law*. A law specifies what should be done when an actor sends and receives a message. Unlike Synchronizers/RTsynchronizers' constraints, a policy has a purely local effect that allows actors to be

subject to multiple coordination policies without interference. However, each policy requires a centralized server where to persists its law and list of members. After a member joins a policy, its interaction with other members is direct, and the law is enforced locally.

Actors with Temporal Constraints (ACT) [42] proposes a new timed actor model based on *active* and *passive temporal constraints*. The former checks that one or more message invocations must occur in a given time interval. The latter checks if one or more message invocations do not occur before a given time interval. Like RTSynchronisers, temporal constraints are expressed as *message patterns*, and they match at most one message. However, ACT's patterns do not support predicate expressions to filter messages by their values. Furthermore, the body of a pattern must be specified inside the handler of a particular message.

Actors-Roles-Coordinators (ARC) [53] proposes a decentralized three-layer coordination model based on message manipulation. Actors (*bottom layer*) represents the computational entities of a system. They are oblivious to the coordination constraints enforced by the *meta-actors* of the upper layers. Roles (*middle layer*) represent static abstractions for the coordinated behaviours shared by a group of actors. This layer observes and manipulates messages of the actors playing their roles (a.k.a. intra-role coordination). In this way, ARC decouples the syntactic dependencies between the actors and coordinators. Coordinators (*top layer*) are responsible for the coordination of different roles (a.k.a. inter-role coordination) by imposing determined constraints. The ARC model was extended in [68] with the semantics of two new coordination operators, i.e., `precede` and `select` to express temporal and spacial coordination constraints, respectively. The former sets a quantitative temporal order among two or more messages to enforce intra-role coordination. Like RTSynchronizers, the `precede` primitive assumes that all the actors share the same global wall-clock time. The latter primitive describes inter-role coordination constraints to select a group of actors to which a message can be dispatched. Although the `select` primitive can define generic patterns (e.g., all lights in room X), they can only match conjunctions of individual messages.

Scoped-Synchronizers[19] tries to mitigate bad intentional constraints imposed by malicious actors over others, which results in a denial of service at the targets. Unlike the original proposal [24], the scoped-synchronizers constrain the sender of a message instead of its receiver. Furthermore,

Listing 3.2 Example of sequencing control in Ambient Contracts

```
1  def SeqProtocol := MessageProtocol: {
2    def init() {
3      (on: "login") => { next() };
4    };
5
6    def next() {
7      (on: "logout") => { end() };
8      (on: "buy") => { next() };
9    };
10
11   def end() { };
12   def start() {
13     init();
14   };
15 };
16
17 def Interface := object: {
18   def seq := provide: seq
19              withContract: any -ensure_m(SeqProtocol)-> any;
20 };
21
22 def buyforme(user,item) {
23   o<-login(user);
24   o<-buy(item);
25   o<-logout(user);
26 };
```

such constraints only will take effect if the actor enacting the synchronizer holds the *synchronization capability* to the message source (*sender-actor*). Synchronization capabilities are used as a scope mechanism to restrict the effect of synchronization constraints to a subset of messages. However, they can not completely prevent deadlocks that may arise from incompatible overlapping constraints.

Contracts [66] allow developers to monitor the outgoing messages and implement protocols to check which messages are being sent using a single language (AmbientTalk [82]). Listing 3.2 shows an example of a sequence protocol in Ambient Contracts copied from [66]. This protocol defines the valid transition of states required to buy a product (lines 1-15). In lines

17-20 we observe how developers can export their protocols. Finally, lines 22-26 illustrate an example of a `buyforme` function that satisfies the above protocol.

Multiparty Session Actors (MSA) [46] proposes a specification and runtime-verification framework based on multiparty session types [36] to ensure correct sequencing of actor interactions. In this framework, actors may play multiple roles and can participate in multiple sessions simultaneously. Safety interaction protocols between actors are defined using a declarative protocol description language, called Scribble [74]. The projection of such protocols to actors is made using annotations in their definition. This adaptation of [36] allows actors to verify that the attributes of a message match their specified types, and the overall order of interactions is correct. These communication constraints are checked dynamically by compiling of Scribble protocols into finite state machines.

MSA-Erlang [23] presents a library for runtime monitoring of Erlang applications based on MSA [46]. This library provides developers with a new Erlang behaviour (`ssa_gen_server`) to monitor the communication between actors. Like in [46], MSA-Erlang's application-level protocols among communicating actors are described using Scribble and are enforced using finite-state machines.

lchannels [64] introduce a session-based programming model based on Scala types. This lightweight model allows actors to verify their communication protocols by representing session types as Scala types, instead of requiring language extensions or a protocol description language (e.g., Scribble). Furthermore, this model provides support for local and distributed communication as a generalization of continuation-passing style protocols. lchannels help the Scala compiler to detect out-of-protocols messages before a session starts, with the exception of linearity errors (e.g., originated by futures), which are checked at runtime.

Effpi [65] extends lchannels and presents a novel compile-time message verification tool. This new tool is implemented as an embedded domain-specific language in Dotty (a.k.a. Scala 3). It verifies safety and liveness properties of actor-based programs via type-level model checking. Furthermore, it allows actors to enforce a particular order of outgoing messages through its sequencing operator (»).

OTyPe [33] adds support for runtime verification of incoming messages of actors based on Erlang's behaviours [1]. A behaviour (e.g., gen_server)

resembles an interface commonly found on object-oriented languages. Erlang developers use behaviours to divide the code for an actor in a generic part (*a behaviour module*) and a specific part (*a callback module*). OTyPe uses a combination of type inference and automatic code injection techniques to discards malformed/ill-typed messages received by a behaviour before calling its callback function. Furthermore, it can check types of guard expressions constraining a message pattern handled by a behaviour's callback.

Overall, synchronization abstractions of Monitor & Verification (MV) proposals can filter messages based on their values and time constraints (e.g., [52, 42, 68]). However, [46, 33] only supports type-based constraints. In this category, only [52] can detect the absence of messages, but it assumes that actors have their local clocks synchronized, and their invocations are scheduled atomically. Like in traditional actor-based languages, the synchronization abstractions of these proposals always match the *oldest* message in the actor's inbox. Table 3.3 summarizes the supported synchronization requirements defined in section 2.3 found on MV proposals.

3.1.4 Local Synchronization

The synchronization abstractions of the proposals described in this category are executed locally in the actor, unlike the ones described in sections 3.1.2 and 3.1.3. We can distinguish two main groups in this category. The first one targets synchronization abstractions based on promises, futures, or message-passing continuations [25, 85, 50]. These proposals allow the synchronization and chaining of responses to individual message invocations. The second group extends the actor interface to match multiple messages. Proposals in this group [32, 69, 49, 47] expand the matching capabilities of the traditional actor's `receive` primitive with *join patterns* [12]. Join patterns (joins for short) consist of two main components: a *conjunction of message patterns* to match, and a *body* or *reaction* that will be executed after a full match. A full match occurs when there is a match of all required messages by the join pattern. In the rest of this section, we briefly describe each of the above proposals and summarize their support for the synchronization primitives identified in section 2.3.

Activators [25] proposes two message synchronization abstractions called *activators* and *receptionists*. On the one hand, activators are patterns defined in an actor that wait for a particular set of messages before

Table 3.3 Synchronization requirements supported by MV proposals

	Filter		**Selection**	**Correlation**			**Accum.**		**Transf.**
	Content-based	Time-based	Flexible	Conjunction	Disjunction	Sequencing	Count-based	Time-based	Aggregation
Synchronizers [24]	x	-	-	x	x	-	-	-	-
RTSynchronizers [52]	x	x	-	x	x	x	-	-	-
Scoped-Synchronizers [19]	x	-	-	x	x	-	-	-	-
Moses [45]	x	-	-	-	-	-	-	-	-
ATC [42]	x	x	-	-	x	-	-	-	-
ARC [53, 68]	x	x	-	-	-	x	-	-	-
Ambient Contracts [66]	x	-	-	x[a]	-	x	-	-	-
MSA [46]	x[b]	-	-	-	-	-	-	-	-
MPSA-Erlang[23]	x	-	-	-	-	-	-	-	-
lchanels [64]	x[c]	-	-	-	-	-	-	-	-
Effpi [65]	x[c]	-	-	-	-	x[d]	-	-	-
OTyPe [33]	x[b]	-	-	-	-	-	-	-	-

[a] Conjunction is only enforced to invocation of messages
[b] Additional type-based constraints are applied to message's attributes (e.g., MsgA(int, string))
[c] Type constraints to outgoing messages (object-level) are checked during the compilation phase
[d] Sequencing is only enforced to outgoing messages

triggering their continuation (a.k.a. reaction). They can be defined as a conjunction or disjunction of messages which are retrieved from a group of special actors called receptionists. Furthermore, activators can constraint the potential messages based on a predicate expression before remove them from the receptionists. On the other hand, a receptionist only acts as a handler of messages that can be monitored by multiple actors. Multiple activators can be waiting for messages from the same receptionist. However, a message can only be consumed by one activator. Similarly, a message can satisfy multiple activators of an actor, but only one is activated. In both cases, the selection is non-deterministic, and it is the responsibility of the developer to implement a selection policy for each case.

SALSA [85] introduces three abstractions to coordinate interaction among a group of actors. These abstractions base their synchronization properties on a particular return value, called *token*. The first abstraction, *token-passing continuations*, allows actors to synchronize and chain individ-

Listing 3.3 Example of the zip operator of Reactive Isolates

```
1  events onCase {
2    case Login(u) =>
3      val ec = keyCenter ? GetCert(u)
4      val ea = authServer ? GetAuth(u)
5      (ec zip ea) onCase {
6        case (cert, auth) =>
7          tokens(user) = compute(cert, auth)
8          user ! tokens(user)
9      }
10 }
```

ual message invocations in a similar way to the pipe command from UNIX[2]. In contrast, the second one, j*oins continuations* allow actors to receive a list of tokens returned by multiple actors once they have all finished processing their respective messages. Join continuations do not enforce a particular order for the tokens' resolution neither a time window for their resolution. The last abstraction, *first-class continuations*, enable actors to delegate a computation to another actor in the system.

Reactive Isolates (RI) [50] proposes a novel concurrency model to overcome the lack of message protocol composition of the actor model. This model extends the traditional actor model in three ways. First, isolates (actors) can define multiple message entry points, called *channels*. Channels are first-class objects that allow developers to declare separated protocol messages instead of defining them in a single-message handling construct (e.g., Erlang's `receive` primitive). Second, RI provides a `zip` abstraction to wait for a particular combination of messages (a.k.a. multi-party protocols). This abstraction does not extend the interface of an actor. Instead, it composes the response of individual asynchronous messages using futures into a new complex event that is put on the actor's inbox. Listing 3.3 illustrates how developers can use the `zip` operator (line 5) to compose the *response* of individual asynchronous messages (lines 3, 4) into a newly composed message (line 7) which is then put on the actor's inbox (line 8). The example shown in this listing is copied from [50].

Scala Joins [32] introduce a join pattern library for synchronizing multiple synchronous and asynchronous messages in both actors and threads

[2]Pipe command - https://en.wikipedia.org/wiki/Pipeline_(Unix)

Listing 3.4 Example of join pattern in JErlang

```
1  receive
2   {reindeer, Pid1} and {reindeer, Pid2} and {reindeer, Pid3} and
3   {reindeer, Pid4} and {reindeer, Pid5} and {reindeer, Pid6} and
4   {reindeer, Pid7} and {reindeer, Pid8} and {reindeer, Pid9} ->
5     % deliver presents
6     [Pid1, Pid2, Pid3, Pid4, Pid5, Pid6, Pid7, Pid8, Pid9];
7
8    {elf, Pid1} and {elf, Pid2} and {elf, Pid3} ->
9     % discuss R&D possibilities
10    [Pid1, Pid2, Pid3]
11 end
```

based systems. This library embraces an extensible message pattern-matching approach based on extractors (partial functions). Additionally, developers can specify more complex message constraints using predicate expressions called guards. However, Scala Joins only supports conjunctions of individual messages. Furthermore, joins' body reactions are coupled with the definition of their respective joins.

JErlang [49] extends Erlang to bring support for joins patterns. JErlang's joins target the synchronization of both synchronous and asynchronous messages. Like in [32], joins can only express conjunctions of single messages, and they always match the oldest relevant message in the actor's inbox. A relevant message is one that satisfied the constraints of a message pattern or in this case a join pattern. Additionally, a message can be consumed by multiple joins by using the optional *propagation* property in the definition of a join. Listing 3.4 shows a snippet of the JErlang solution to the well-known Santa Claus problem [80]. This solution is copied from [49]. In this synchronization problem, Santa sleeps at the North Pole waiting to be awakened by nine reindeer or three elves to execute some work with them. However, the waiting group of the former has higher priority if both full groups gather at the same time. Listing 3.4 shows that the `receive` primitive is now able to match a conjunction of messages instead of individual messages. For example, lines 2-4 define a join pattern that will match nine `reindeer` messages. In contrast, the join pattern defined in line 8 will match three `elf` messages.

JCThorn [47] extends the Thorn language with constructs based on the join-calculus. This extension provides two variations of the same synchronization abstraction named *joins* and *chords*. The former represents the low-level communication mode and they resemble JErlang's joins. The latter is a more high-level abstraction that is translated to joins at runtime. Like JErlang, JCThorn supports the synchronization of both synchronous and asynchronous messages. Furthermore, developers can assign numeric priorities to its joins. This feature enforces that messages in the actor's inbox have to be checked against a higher priority join before attempting to match a lower priority one.

Multi-headed Message Receive Patterns (MMRP) [69] introduce an extension of an Erlang-like actor model with `receive` clauses containing *multi-headed message patterns*. This kind of patterns resembles the join patterns of JErlang. However, they support multiple matching semantics: *first-match* (like in JErlang) and *rule priority-match* (like in JCThorn).

Overall, join pattern languages support advanced pattern-matching techniques to compact filtering expressions. However, only JErlang, JCThorn, and MMRP support the unification of message patterns' attributes across a join. This mechanism allows join patterns to synchronize the values of shared attributes among multiple message patterns without guard expressions. The join patterns of the above join languages support conjunction of messages. However, only Activators' joins can define both conjunctions and disjunctions with the constraint that they can not be mixed in a join. The proposals analyzed in this section do not enforce a particular matching order of their constituents' messages. Furthermore, despite the expressiveness of their abstractions, actor-based join languages force developers to statically bind join patterns and reactions during the definition of the former. Furthermore, they lack support for time-based constraints and accumulation abstractions. We can observe this last limitation in JErlang's join patterns (see listing 3.4 lines 2-4). In this solution, the size of the reindeer join grows proportionally to the number of reindeer (messages) being synchronized. Table 3.4 summarizes the supported synchronization requirements supported by local synchronization (LS) proposals.

In this section we have observed that different extensions to the traditional actor model presented until now fall short to support the synchronization requirements identified in section 2.3. In order to support such

Table 3.4 Synchronization requirements supported by LS proposals

	Filter		Selection	Correlation			Accum.		Transf.
	Content-based	Time-based	Flexible	Conjunction	Disjunction	Sequencing	Count-based	Time-based	Aggregation
Activators [25]	x	-	-	x[a]	x	-	-	-	-
Salsa [85]	x	-	-	x[a]	-	-	-	-	-
Reactive Isolates [50]	x	-	-	x[a]	-	-	-	-	x
Scala Joins [32]	x	-	-	x	-	-	-	-	-
JErlang [49]	x	-	-	x	-	-	-	-	-
JCThorn [47]	x	-	-	x	-	-	-	-	-
MMRP [69]	x	-	-	x	-	-	-	-	-

[a] Conjunction is only enforced to invocation of messages

synchronization abstractions in an actor-based language, we have gotten our inspiration from CEP systems. Hence, in the next section, we will look at types of synchronization mechanisms that exist for CEP and how we can integrate them into an extension of the actor model.

3.2 Complex Event Processing

In this section, we analyze the support of the synchronization requirements identified in section 2.3 by complex event processing (CEP) systems/languages. CEP systems, unlike the above proposals have been designed for processing large amounts of data. These systems focus on the detection of complex patterns of correlated temporal events instead of filtering individual events. Hence, CEP systems seem like a normal fit to tackle the type of synchronizations that we envision. Unfortunately, the technical complexity of these systems is higher than in join-pattern languages. They typically require knowledge of a stack of various technologies from developers for their deployment. Furthermore, CEP *libraries* frequently lack expressive APIs since they rely on the programming abstractions of its host language. In contrast, CEP *languages* provide greater expressiveness, but they mostly focus on the detection of events using a SQL-like syntax. They also lack support for the *unification* of multiple events. Nevertheless, as we will see, they offer a great source of inspiration to build an enhanced actor model

Table 3.5 Synchronization requirements supported by CEP proposals

	Filter		**Selection**	**Correlation**			**Accum.**		**Transf.**
	Content-based	Time-based	Flexible	Conjunction	Disjunction	Sequencing	Count-based	Time-based	Aggregation
Amit [5]	x	x	x	x	-	x	x	-	-
CEDGME [48]	x	x	-	x	x	x	-	-	-
PADRES [43]	x	x	-	x	x	x[a]	-	-	-
CEDR [11]	x	x	x	x	x	x	x	-	-
SASE [86]	x	x	x	-	-	x	-	-	-
CAYUGA [18]	x	x	-	-	-	x	-	x	x
SASE+ [31]	x	x	x	-	-	x	-	x	x
PB-CED [8]	x	x[a]	-	x	x	x	-	-	-
EventJava [20]	x	x[a]	-	-	-	x[a]	x	-	x[b]
RACED [14]	x	x	x	x	x	x[a]	-	-	-
ZStream [44]	x	x	x	x	x	x	x	-	x
TESLA [15]	x	x	x	-	-	x	-	x	x
ETALIS [9]	x	x	x	x	x	x	-	x	x
PARTE [54]	x	x	-	x	-	x	-	-	-

[a] The developer has the responsibility to do this action manually using the timestamp attribute

[b] Specific aggregation abstractions (e.g., `AVG`) are not built-in; instead, developers must use standard iteration constructs (e.g., `for`) to aggregate values

that is capable of delivering with the kinds of synchronization that we are after.

Table 3.5 summarize the results of our analysis. Although the analyzed systems/languages are academic proposals, their abstractions are equal or more expressive than modern CEP applied in the industry (e.g., Flink-CEP [70] and Esper [38]). We can observe at first sight that CEP systems have better overall support for each synchronization abstraction identified in section 2.3. However, their content-based filters are mostly based on standard boolean predicates, lacking of advanced pattern-matching techniques. Only [86, 11, 31] support a basic pattern-matching mechanism called *equivalence/equality* tests. These predicates enforce that *shared* attributes across an entire event sequence must have the same, different, or a particular value. Listing 3.5 shows an equivalence test example in CEDR [11]. Line 4 establishes that all the constituents events of `EVENT_PATTERN` share the same `machine_id`. This example also shows how the detection of a particular

Listing 3.5 Equivalence test example in CEDR

```
1 Q1: FAILED_UPGRADE
2 EVENT_PATTERN SEQUENCE(INSTALL x, SHUTDOWN y,
3                        NOT (RESTART z, WITHIN 5 minutes))
4 WHERE CorrelationKey(machine_id, Equal)
5 NOTIFY Each x, First y
```

sequence of events can be enforced. In this case, the query `FAILED_UPGRADE` (line 1) requires that an event `INSTALL` must be followed by a `SHUTDOWN` event which cannot be followed by a `RESTART` event within the next 5 minutes (lines 2-3).

Unlike join languages, CEP systems strongly support time-based predicates (see column *Time-based* in table 3.5). However, not all of them implement high-level time filtering abstractions like CEDR's `not` and `within`. For example, listing 3.6 showcases how EventJava only makes accessible implicit time attributes to developers, and they have to filter the relevant messages manually (line 7). Although CEP systems support a rich set of timing abstractions, negation is sometimes limited to a sequence of events (e.g., SASE, SASE+).

Listing 3.6 Manual timing constraint example in EventJava

```
1 event
2   tvRelease(String model, float price, String date),
3   tvReview[5](String model, File review, float rating)
4 when(
5   for i in 0..3 tvReview[i].model == tvReview[i+1].model &&
6   for i in 0..4 tvReview[i].rating >= 3.5 &&
7     tvReview[4].time - tvReview[0].time = 30*24*60*60*1000 &&
8     tvReview[0].model == tvRelease.model
9 ) {...}
```

Time-based predicates in CEP systems focus at two levels: event and pattern. The former allows developers to filter *individual* events of a pattern. The latter sets a *global* time constraint to the whole pattern. Systems like [5, 48, 11, 15, 54] support advanced timing constraints for both levels. For example, in TESLA (see listing 3.7), developers can specify specific timing constraints between different events of a pattern (line 5). At the same time, other solutions such as [86, 18, 31, 20, 8, 14, 44, 9] only

Listing 3.7 Advanced timing constraint example in TESLA

```
1 define Fire(Val)
2 from Smoke(Area= $x) and
3      each Temp(Val > 45 and Area= $x)
4      within 5 min from Smoke
5 where Val=Temp.Val
```

support one of the two types of time constraint levels. In the particular example of listing 3.7, a fire event will be created when the temperature of an area is higher than 45 degrees and some smoke is detected in the same area within 5 min.

CEP systems also have better support for correlation abstractions (see column *Correlation*) than join-pattern approaches. However, only [48, 8, 14, 11, 44, 9] support all three types of correlations. Developers of systems like [43, 20, 14], have to manually set the desired sequence order of the events using their timestamp attributes like in listing 3.6 (line 7). Furthermore, the analyzed CEP systems lack support for accumulation abstractions (see column *Accum.*). As we observe, frequently, only one of both abstraction is supported. Moreover, computed aggregated values can not be included in their pattern conditions (see column *Aggregation*), except for [18, 31, 44, 15, 49, 9]. For example, EventJava does not provide high-level aggregation abstractions (e.g., `AVG`) Instead, it gives developers a generic iterator abstraction as shown in listing 3.6 (lines 5-6).

Table 3.6 summarizes the support for the synchronization requirements identified in Section 2.3 by the state-of-the-art technologies analyzed in this chapter.

3.3 Conclusion

The traditional actor model offers basic abstractions to synchronize messages within a group of actors.

Although extensions like join pattern improve the synchronization capabilities of canonical actor-based languages, they still fall short to support the requirements identified in section 2.3. The analyzed join extensions mostly support conjunctions of individual messages. Disjunctions have to manually defined as new joins, leading to duplicated code. Furthermore,

Table 3.6 Synchronization requirements supported by state-of-the-art actor-based languages/frameworks and CEP systems

	Filter		**Selection**	**Correlation**			**Accum.**		**Transf.**
	Content-based	Time-based	Flexible	Conjunction	Disjunction	Sequencing	Count-based	Time-based	Aggregation
The canonical actor model									
PLASMA [34]	x	-	-	-	-	-	-	-	-
ABCL/1 [87]	x	-	-	-	-	-	-	-	-
Rossete [79]	x	-	-	-	-	-	-	-	-
Erlang [73]	x	-	-	-	-	-	-	-	-
Elixir [76]	x	-	-	-	-	-	-	-	-
Scala/Akka [75]	x	-	-	x[a]	-	-	-	-	-
AmbientTalk [82]	x	-	-	x[a]	-	-	-	-	-
Pony [13]	x	-	-	-	-	-	-	-	-
Communication model extensions									
ActorSpace [7]	x	-	-	-	-	-	-	-	-
TOTAM [67]	x	-	-	-	-	-	-	-	-
Directors [84]	x	-	-	-	-	-	-	-	-
Syndicate [27]	x	-	-	-	-	-	-	-	-
AErlang [17]	x	-	-	x[a]	-	-	-	-	-
interActors [29]	x	-	-	x[a]	-	-	-	-	-
RCF [30]	x	-	-	-	-	-	-	-	-
Coordinators [40]	x	-	-	-	-	-	-	-	x
Monitor & Verification									
Synchronizers [24]	x	-	-	x	x	-	-	-	-
RTSynchronizers [52]	x	x	-	x	x	x	-	-	-
Scoped-Synchronizers [19]	x	-	-	x	x	-	-	-	-
Moses [45]	x	-	-	-	-	-	-	-	-
ATC [42]	x	x	-	-	x	-	-	-	-
ARC [53, 68]	x	x	-	-	-	x	-	-	-
Ambient Contracts [66]	x	-	-	x[a]	-	x	-	-	-
MSA [46]	x[b]	-	-	-	-	-	-	-	-
MPSA-Erlang [23]	x	-	-	-	-	-	-	-	-
lchanels [64]	x[c]	-	-	-	-	-	-	-	-
Effpi [65]	x[c]	-	-	-	-	x[d]	-	-	-
OTyPe [33]	x[b]	-	-	-	-	-	-	-	-
Local synchronization									
Activators [25]	x	-	-	x[a]	x	-	-	-	-
Salsa [85]	x	-	-	x[a]	-	-	-	-	-
Reactive Isolates [50]	x	-	-	x[a]	-	-	-	-	x
Scala Joins [32]	x	-	-	x	-	-	-	-	-
JErlang [49]	x	-	-	x	-	-	-	-	-
JCThorn [47]	x	-	-	x	-	-	-	-	-
MMRP [69]	x	-	-	x	-	-	-	-	-
Amit [5]	x	x	x	x	-	x	x	-	-
CEDGME [48]	x	x	-	x	x	x	-	-	-
PADRES [43]	x	x	-	x	x	x[a]	-	-	-
CEDR [11]	x	x	x	x	x	x	x	-	-
SASE [86]	x	x	x	-	-	x	-	-	-
CAYUGA [18]	x	x	-	-	-	x	-	x	x
SASE+ [31]	x	x	x	-	-	x	-	x	x
PB-CED [8]	x	x[a]	-	x	x	x	-	-	-
EventJava [20]	x	x[a]	-	-	-	x[a]	x	-	x[b]
RACED [14]	x	x	x	x	x	x[a]	-	-	-
ZStream [44]	x	x	x	x	x	x	x	-	x
TESLA [15]	x	x	x	-	-	x	-	x	x
ETALIS [9]	x	x	x	x	x	x	-	x	x
PARTE [54]	x	x	-	x	-	x	-	-	-

[a] Conjunction is only enforced to invocation of messages
[b] Additional type-based constraints are applied to message's attributes
[c] Type constraints to outgoing messages are checked during the compilation phase
[d] Sequencing is only enforced to outgoing messages

their patterns lack time constraints and order matching. However, joins provide a more elegant and local solution to improve the expressiveness of an actor language.

We explored the synchronization mechanism available in CEP systems/languages. Our study found that proposals in this domain allow developers to define complex synchronization patterns with flexible selection and consumption event policies. Unlike join languages, CEP proposals target systems with larger volumes of messages. Nevertheless, they do not support advanced pattern-matching techniques to filter events like joins languages. CEP systems also restrict their reactions to the emission of a single composite event, except for EventJava and PARTE. Moreover, CEP languages tend to be declarative non-turing-complete languages which also lack support for the unification of multiple events. In contrast, join languages offer more powerful reactions options since they have full access to the capabilities of their host languages (e.g., they can spawn new actors or send multiple messages).

We conclude that a fusion between join-patterns and CEP features could be the sweet spot between the two approaches for tackling the synchronization requirements presented in section 2.3. In the next chapter, we explore the reconciliation of join patterns and complex event processing techniques into an actor language.

CHAPTER 4

Sparrow: A DSL for Actor Coordination

In this chapter, we introduce the main contribution of this dissertation, to wit a novel join pattern language called Sparrow. Like previous join pattern languages [32, 49, 47], Sparrow extends the traditional *single-message* match interface of actors [35] to support *multiple-message* match. CEP languages and frameworks heavily inspire Sparrow's join pattern language design. Sparrow has been implemented as a domain-specific language (DSL) in Elixir that relies heavily on Elixir's macro facilities for its implementation. As it is implemented as a DSL, it has access to all of the programming language constructs of its host language. As such, Elixir syntax is an integral part of the Sparrow language. To ensure that this dissertation is self-contained, in the next section, we will briefly introduce Elixir.

4.1 Elixir in a Nutshell

This section introduces basic concepts of Elixir to facilitate the understanding of Sparrow. Like Erlang, Elixir is a dynamically typed actor-based language that runs on top of the BEAM[1] virtual machine. Furthermore, it embraces Erlang's message-passing actor model [35] to build scalable, fault-tolerant, and distributed systems. However, Elixir was designed to

[1]BEAM - Virtual machine that executes user code in the Erlang Runtime System

Listing 4.1 A counter actor in Elixir

```
1  defmodule Counter do
2
3    def start() do
4      spawn fn -> listen(0) end
5    end
6
7    def listen(count) do
8      receive do
9        :inc ->
10         listen(count + 1)
11       {:val, sender} ->
12         send sender, count
13         listen(count)
14     end
15   end
16
17 end
```

be an extensible language through metaprogramming (see section 6.1). For instance, Elixir has a macro system inspired by Lisp-like languages.

Listing 4.1 shows the definition of a `Counter` actor and the essential constituents of a typical Elixir program. Like any other Elixir program, the actor definition is encapsulated in a *module*. As its name (`Counter`) implies, this actor increments a value by one, and it can also return the current value of its counter. Both actions are triggered by messages which constitute the *actor's interface*. The interface of an actor is defined by the set of *message patterns* it understands and can react to. In Elixir, similar to Erlang, this is done using a `receive` statement (lines 8-14). A common approach is to use a recursive function to receive messages in a loop. When a message is sent to an actor, the message is stored in the actor's inbox. The `receive` block goes through the current actor's inbox searching for a message that matches any given patterns.

The recursive function `listen` (lines 7-15) takes a single argument which represents the current value of the counter and at the same time the *actor's state*. Its body defines the actor's interface that consists of two

patterns. The first one matches messages which values is the atom[2] `:inc`. This pattern is used to increase the current value of the counter (lines 9-10). The second one instead matches a *tuple* where the first element identifies the message's type (`:val`), while the second element represents the sender's id (`sender`). This last pattern introduces the primitive `send` which is used to return the current value of the counter to the sender (lines 11-13). In both cases, the two patterns end with a *tail recursive* call of the `listen` function (lines 10, 13) to read the next message of the actor's inbox. If there are no new messages, the actor waits indefinitely for a new message to arrive. This waiting behaviour puts the actor in a suspended state that does not waste CPU cycles.

The function `start` (lines 3-5) is used to create a new actor using the `spawn` primitive. This primitive takes an anonymous function that will perform the initial call to the `listen` function, thereby launching the actor's tail recursive process.

Like in Erlang, the actor's inbox is a FIFO queue limited only by the available computer's memory. The receiver actor consumes messages in the order in which they were received. A message can be removed from its queue only if it's matched (*consumed*) by one of the `receive`'s patterns. Furthermore, the matching process inside the receive primitive occurs from top to bottom. The interested reader can find detailed documentation of Elixir in [76].

4.2 Sparrow by Example

Before detailing Sparrow's language abstractions, this section will give a sneak preview of how developers can program the coordination logic of scenarios 1 and 2 (see section 2.1) in Sparrow.

4.2.1 Enhanced Actors

Listing 4.2 shows the definition of an actor `LightManager` whose purpose is to turn on/off the light of any room if some conditions are met. In Sparrow, the interface of an actor is defined by a set of patterns using the `pattern` primitive. This primitive uses a *tuple* to specify the type of

[2] Atoms are constants whose values are their own name.

Listing 4.2 A solution to an instance to scenarios 1 and 2 in Sparrow

```
1  defmodule LightManager do
2      use Sparrow.Actor
3
4      pattern motion as {:motion, id, :on, room}
5      pattern light as {:light, id, status, room}
6      pattern low_light as {:amb_light, id, illuminance, room}
7                           when illuminance < 40
8
9      pattern on_motion as motion
10                         and light{status= :off}
11                         and low_light,
12                         options: [last: true]
13
14     pattern no_motion as not motion[window: {2, :mins}]
15                         and light{status= :on},
16                         options: [last: true]
17
18     reaction turn_on_light(l, i, t), do: # send on command
19     reaction turn_off_light(l, i, t), do: # send off command
20
21     react_to on_motion, with: turn_on_light
22     react_to no_motion, with: turn_off_light
23
24 end
```

relevant[3] messages for it. The first element of that tuple represents the *message's type*, and the rest represent the *message's attributes*, which can be set to a particular value. The actor `LightManager` declares five patterns. The first three (lines 4-7) are used as *base* patterns to compose the patterns that will specify the logic to turn on/off the lights of a particular room (lines 9-16).

- The `motion` pattern (line 4) will match a `:motion` message if the value of its second attribute (`location`) is `:on`. The rest of the message's attributes (`id` and `room`) do not specify any constraints on their values.

[3] A *relevant message* is a message that satisfies the constraints of a pattern.

- The `light` pattern (line 5) will match a `:light` message without any constraints on that message attributes' values.
- The `low_light` pattern (lines 6-7) will match `:amb_light` messages if their `illuminance` value is less than 40 lux. The rest of the message's attributes do not specify any constraints on their values.
- The `on_motion` pattern (lines 9-12) uses a conjunction between the above base patterns to determine if the light of a particular room should be turned on. However, it will only accept a message from the pattern `light` if the `status`'s value of that message is `off` (line 10). Additionally, this pattern always requires to match the most recent relevant messages in the actor's inbox (line 12).
- The `no_motion` pattern (lines 14-16) uses a conjunction between the base patterns `motion` and `low_light` to determine if the light of a particular room should be turned off. In this case, it does not want to receive any message from the `motion` pattern in a time window of 2 minutes (see line 14). Furthermore, `no_motion` will only accept messages from the pattern `light` if the `status`'s value of its messages is `on` (line 115). Finally, like the `on_motion` pattern, it always requires to match the most recent relevant messages in the actor's inbox (line 16).

The primitive `pattern` allows developers to define a fully functional actor's interface, but no action will be executed upon a *full match* of one of its patterns. *A pattern has a full match if all its conditions are met.* In Sparrow, *reactions* define the logic to execute upon a full match of a pattern. The process of executing the reactions of a pattern is called *pattern activation*. Lines 18-19 show the declaration of two reactions: `turn_on_light` and `turn_off_light`. For clarity, the body of both reactions was omitted. Comments with their description were used instead. These reactions are attached to their respective patterns using the function `react_to` (lines 21-21). In the rest of this dissertation, will we refer to the action of executing the reactions of a pattern as *pattern activation*.

In a nutshell, the definition of a Sparrow actor consists of four parts:

- Import Sparrow's language abstractions (line 2).
- Define relevant message patterns (lines 4–16)

- Define pattern reactions lines 18-19)
- Bind patterns with their reactions (lines 21-22)

4.2.2 Language Syntax Overview

As we mentioned earlier, Sparrow is implemented as a domain-specific language (DSL) that leverages the macro facilities of Elixir for its implementation. At the same time, this decision *shaped* and *restricted* the expressiveness of its language abstractions. Sparrow's primitives are restricted by the set of valid expressions and operators supported by Elixir's parser. Although Elixir provides developers with a rich macro system, it does not allow redefining used operators (e.g., and, or), or create new ones. However, it allows redefining a list of unused operators supported by its parser (e.g., ~>). Figure 4.1 shows an EBNF-styled syntax definition of Sparrow. In this figure, observe that Sparrow can be subdivided into two smaller languages.

On the one hand, there is *Sparrow's pattern language*. In traditional actor languages with pattern-matching (e.g., Erlang, Elixir), messages are represented by any value, mostly *tuples*. In such languages, *message patterns* usually decompose a single message into its constituents using pattern-matching with logical variables. Sparrow builds on and extends this idea by adding a rich set of additional synchronization operators to its pattern language through the non-terminal `<p-definition>`. This non-terminal introduces `pattern <identifier> as <pattern>` syntax to bind any pattern to a name, where `<identifier>` can be any valid identifier and `<pattern>` can be any valid Sparrow pattern. This special form can be used to abstract and reuse patterns in Sparrow. Section 4.3 will describe in depth the different variants of `<pattern>` and their properties. Briefly, Sparrow's pattern language can be seen as a declarative subset of Sparrow (inspired by CEP languages) that enables the declaration of complex synchronization patterns.

On the other hand, there is *Sparrow's reaction language*. A *pattern reaction* represents the body of a message pattern of traditional actor languages. These languages typically require the programmer to specify the body of a message pattern together with its definition. Sparrow decouples these two definitions to improve the composability and reusability of complex message patterns. Section 4.4 will describe in depth the different

```
(R1) <p-definition>        :=  pattern <identifier> as <pattern>
(R2) <pattern>             :=  <acc-pattern> [<guard>] [, [options: <option>+]]
(R3) <acc-pattern>         :=  <comp-pattern> {|> <transformer>}*
(R4) <comp-pattern>        :=  <elem-pattern> [(and | or ) <elem-pattern>]*
(R5) <elem-pattern>        :=  [not] <selector> [[<operator>+]]
(R6) <selector>            :=  {<symbol>, <attribute>*}
                               | <identifier>[{{<inline-guard>
                               | <alias-op>}+}]
(R7) <attribute>           :=  <value>
                               | <symbol>
                               | <logic-var>
(R8) <guard>               :=  when <expression>
(R9) <inline-guard>        :=  <identifier> = <expression>
(R10) <alias-op>           :=  <identifier> ~> <identifier>
(R11) <symbol>             :=  :<identifier>
(R12) <logic-var>          :=  [(@ | ! )]<identifier>
(R13) <operator>           :=  window: <time>
                               | debounce: <time>
                               | every: <number>
                               | count: <number>
(R14) <transformer>        :=  fold(<expression>, <expression>)
                               | map(<expression>)
                               | bind(<identifier>)
(R15) <option>             :=  seq: <boolean>
                               | interval: <time>
                               | last: <boolean>
(R16) <time>               :=  {<number>, (:secs | :mins | :hours | :days | :weeks)}

(R17) <r-definition>       :=  reaction <identifier> (<arg>, <arg>, <arg>) do <expression> end
(R18) <react-to>           :=  react_to <identifier>, with: <identifier>
(R19) <remove-from>        :=  remove <identifier>, from: <identifier>
(R20) <remove-reactions>   :=  remove_reactions <identifier>

(R21) <arg>                :=  <identifier>
```

Figure 4.1 Sparrow EBNF-styled syntax definition

abstractions defined by the reaction language. Briefly, Sparrow's reaction language can be seen as a superset of the Elixir programming language in which the reaction to a message pattern can be specified.

In the figure 4.1, the definitions of the non-terminals `<expression>`, `<identifier>` `<value>`, `<number>` and `<boolean>` have been omitted and corresponds to ordinary Elixir expressions, identifiers, primitive values, numbers and booleans respectively. A detailed explanation of the grammar rules of both languages will be given in the next sections.

4.3 Sparrow's Pattern Language

Similarly to Elixir, in Sparrow, incoming messages are *pattern matched* against a set of message patterns. When a match is found, the matching messages are *consumed* from the actor's inbox, and the actor starts a process to react to those messages. This reaction is specified in the Sparrow's reaction language (see section 4.4).

Sparrow has support for three types of patterns: *elementary* patterns enable the matching of single messages, *composite* patterns enable the composition of multiple elementary patterns, and *accumulation* patterns enable the accumulation and aggregation of multiple messages of the same type. These patterns incarnate the synchronization requirements identified in section 2.3.

4.3.1 Elementary Patterns

Elementary patterns are the most basic kind of Sparrow's patterns (see figure 4.1 R5). However, they extend the *message patterns* of traditional actor programs (e.g., Erlang, Elixir) in several ways. For example:

- They can be reused to compose more complex patterns.
- They can detect the absence of a certain type of message in a time window.
- They can ignore a certain type of message for some time.
- They can ignore a number n of messages of a certain type.

In the rest of this section, we detail all variant of elementary patterns supported by our DSL.

In Sparrow, developers can define a new pattern using the `pattern` special form (see figure 4.1 R1). Elementary patterns usually start with a *selector* that designates a single message (see figure 4.1 R6). Similar to Elixir's messages, a selector in Sparrow is represented as a tuple. Its first element determines the type of message to match, and it always has a *unique constant* value (e.g., `:window`). The other selector elements are called *attributes* (see figure 4.1 R7), and they can be primitive values (e.g., string, number) or *logic variables*. Logic variables represent a dynamic primitive value that is unknown until the matching of the pattern against a

message that sits in the actor's inbox. Inspired by Elixir's pattern-matching mechanism [81], logic variables of a selector with the same identifier must have the same value. Additionally, an elementary pattern's selector can be accompanied by an optional *operator* and an optional *guard expression*. We will explain these last components in the next sections.

Figure 4.2 shows the definition of an elementary pattern that matches when an open window is detected at any location (this is part of the implementation for scenario 3). The `pattern` keyword is used to give a name to the pattern. In Sparrow, patterns are second-class citizens that can be reused and composed to define complex patterns. The `as` keyword is followed by the *selector* of the pattern. As with plain Elixir, pattern-matching on primitive values can be used to filter messages based on their attribute values. For instance, the pattern shown in this example will only match `:window` messages for which the second attribute (`status`) has as value `:open`. The other attributes (`id, location`) are logic variables and they will match any value.

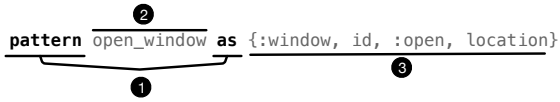


Figure 4.2 Example of an elementary pattern: (1) Primitive used to declare a pattern; (2) Assign a name for future references; (3) Define the pattern's selector

4.3.1.1 Operators

Pattern operators (see figure 4.1 R13) are high-level conditions that further delineate the kind of messages that can be matched by the pattern selector. As we mentioned in section 4.3.1, a selector can optionally be followed by one or more operators enclosed in square brackets. Elementary patterns support three types of operators: *negation*, *debouncing*, and *extensional sequencing*. The integration of Sparrow in Elixir required us to design negation as a *prefix* operator, and debouncing and sequencing as *postfix* operators. In the rest of this section, we will explain in detail each of these operators.

Figure 4.3 shows an implementation for scenario 2 that uses the *negation* operator. For this scenario, we want to turn off the lights after two

minutes without movement. Detecting the absence of motion events can be implemented in Sparrow by using the negation operator (`not`) in combination with a particular time window. The negation operator *must* always be combined with an instance of the `window` operator (e.g., `[window: 2, :mins`). Every time the selector matches a new message, the time of the window operator is reset. Once the time window expires, the pattern is automatically matched.

```
pattern turn_off_light as not {:motion, id, :on, location}[window: {2, :mins}]
                          ❶                                ❷
```

Figure 4.3 Implementation of scenario 2 using a negated pattern: (1) Negate the selector definition; (2) Set the time window

Figure 4.4 shows an implementation of scenario 4 using the *debounce*[4] operator. For this scenario, we only want to match doorbell messages if no other doorbell message was matched in the past 30 seconds. The `debounce` operator does exactly that, the first message that matches the preceding selector automatically matches the entire pattern. Any future messages that follow within the debouncing time are automatically discarded.

```
pattern doorbell_alert as {:doorbell, id}[debounce: {30, :secs}]
```

Figure 4.4 Implementation of scenario 4 using a debouncing time between messages

Figure 4.5 shows a pattern where we are interested in matching every third *heating failure* event. This can be done in Sparrow by following the pattern selector with an *extensional sequencing* operator using the `every` keyword. In our example, only every third message is matched and consumed. All other messages that match the selector are discarded. This pattern can be used as a partial solution to scenario 7 (see section 2.1).

4.3.1.2 Guards

Pattern guards (see figure 4.1 R8) are boolean predicates that are executed after a match of a selector and its operators is found. They augment the

[4]The term `debouncing` is taken from the domain of electrical circuits where a particular debouncing algorithm is used to avoid multiple triggers (within a period) to produce an undesired control output [26].

```
pattern heating_failure as {:heating_f, id, code}[every: 3]
```

Figure 4.5 Example using the extensional sequencing operator (`every`) of Sparrow

pattern-matching of selectors with more complex conditions. Like with traditional Elixir guards [72], Sparrow's guards are only allowed to contain a set of boolean predicate expressions that can always be executed in constant time and are side-effect free. This is a deliberate choice. In this way, Sparrow can make sure that nothing bad happens while executing guards. It also allows Sparrow's macros to optimize the code related to guards efficiently. If a not allowed expression (e.g., a user custom function) is used the compilation process of the actor will fail and an error will be raised. Figure 4.6 shows a guarded pattern that will be activated if it matches an `:open` window message from the `:bedroom` or the `:kitchen`. The full pattern is only matched (and the corresponding reaction fired) when the guard expression evaluates to `true`. If the guard expression evaluates to `false`, the pattern is not matched, and the messages are not consumed.

```
pattern open_window as {:window, id, :open, location}
                    when location == :bedroom or location == :kitchen
```

Figure 4.6 Example of a pattern with a guard expression

Guards can also constrain attributes of patterns composed by multiple selectors (see section 4.3.2). In such patterns, it is also possible to declare a guard for a particular selector. This type of guard is called, *inline guards* Unlike regular guards, inline guards allow developers to write *compact* guard expressions that affect a *single* selector. Except for this last constraint, an inline guard behaves as a regular guard. In the next section, an example of inline guards will be given.

4.3.2 Composite Patterns

Unlike message patterns of traditional actor-based languages (e.g., Scala, Elixir), Sparrow's patterns can be reused, further specified or composed with other patterns to declare a more complex coordination logic. This is only possible for patterns that have been given a name through Sparrow's

`pattern as` primitive. The following sections will detail the different forms in which a pattern can be reused and composed.

4.3.2.1 Reusing patterns

Developers can reuse patterns by specifying a *pattern name* instead of a selector when defining a new pattern. An optional set of *inline guards* (see section 4.3.1.2) can follow this name. Pattern reuse is illustrated in figure 4.7, which shows two semantically equivalent variants (B, C) of a pattern that extends another elementary pattern (A). The pattern `open_window` (A) will match messages of type `:window` if the value of its attribute `state` is `:open`. We can observe in this example the use of the inherited Elixir's pattern-matching approach to filter messages using the atoms `:window` (message's type) and `:open`. The pattern `kitchen_open_a` (B) uses a guard expression to further specify that the open window needs to be detected in the kitchen using the equals operator (`==`) and the atom `:kitchen`. In contrast, the pattern `kitchen_open_b` (C) employs an *inline guard* to substitute the logic variable `location` for the atom `:kitchen` using the match operator (`=`). Both patterns (A and B) will match the same events, namely when a window is opened in the kitchen.

```
A pattern open_window as {:window, id, :open, location}

B pattern kitchen_window_a as open_window when location == :kitchen

C pattern kitchen_window_b as open_window{location= :kitchen}
```

Figure 4.7 Example of pattern reuse in Sparrow

4.3.2.2 Composing patterns

Sparrow patterns can be composed by linking multiple patterns using a logic operator. Sparrow supports both conjunctions (`and`) and disjunctions (`or`) of patterns. Like with traditional actor languages; disjunction can also be achieved by separating each of the patterns in a disjunction. However, as Sparrow also has conjunction, which is typically not supported by traditional actor languages, we also syntactically support disjunction. In the same way, Elixir inspired us to add support for unification of logic variables of a selector. Sparrow has support for *unification* of logical

variables crossing the constituents of a composite pattern. For example, figure 4.8 shows a composite pattern that will be activated, if it receives a carbon dioxide (`:co2`) and carbon monoxide (`:co`) message from the same device notifying that certain level (previously specified) was detected (`true`). This last constraint is enforced by the unification of the `id` attribute of both *anonymous* patterns. Anonymous patterns behave like any other pattern, except for the fact that they cannot be reused.

```
pattern carbon_alert as {:co2, id, true} and {:co, id, true}
```

Figure 4.8 Example of logic variable unification in a composite pattern

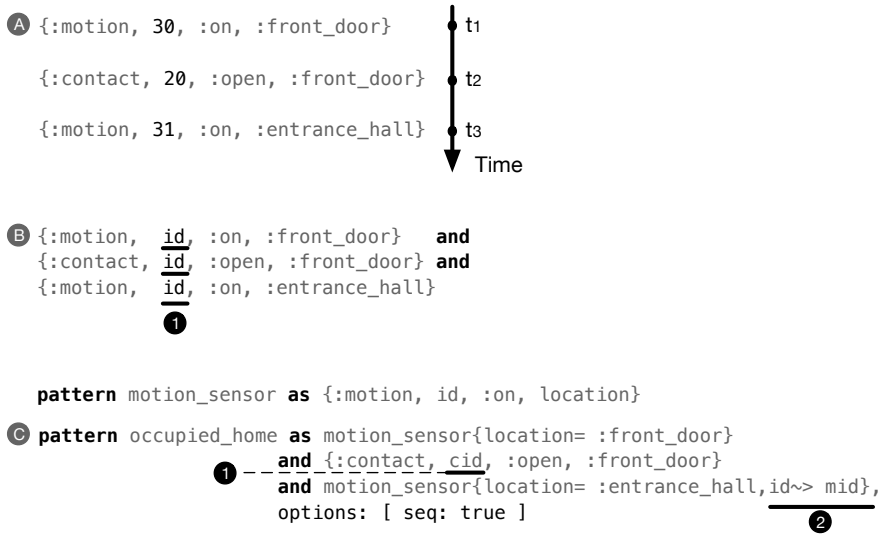
Figure 4.9 illustrates a composite pattern (`occupied_home`) that partially implements the smart home scenario 5 explained in section 2.1. For this scenario, we consider a home inhabited by one person. We are interested in detecting when the person is arriving home by first detecting *motion* at the *front door*, followed by receiving an *open door* event, followed by detecting *motion* in the *entrance hall*. In our example, the `occupied_home` pattern is defined as a conjunction of three patterns. On the one hand, the first and third pattern reuse the `motion_sensor` pattern to match only motion events from the *front door* and *entrance hall* respectively. On the other hand, the second one is an *anonymous* pattern that matches door opening events.

```
pattern motion_sensor as {:motion, id, :on, location}

pattern occupied_home as motion_sensor{location= :front_door}
                      and {:contact, id, :open, :front_door}
                      and motion_sensor{location= :entrance_hall},
                      options: [ seq: true ]
```

Figure 4.9 Example of use of sequencing operator (`seq`)

By default, a composite pattern does not enforce any particular order in which its constituent patterns must match. However, that behaviour can be changed using the *intensional sequencing* operator (`seq`). Due to the syntax limitations imposed by Sparrow's host language (see section 4.2.2), this operator is specified as an entry in the general options of the pattern. In the rest of this dissertation, we will refer to this operator as *sequencing* operator for short. Figure 4.9 showcases the use of this operator to specify that



■ **Figure 4.10** Example of renaming logic variables using the alias operator (~>)

the `occupied_home` pattern can only be matched if the matched messages arrive in the same order as they are specified in its definition.

4.3.2.3 Renaming Logic Variables

As was mentioned in section 4.3.2.2, Sparrow has support for the unification of logical variables crossing the constituents of a composite pattern. This is often desirable as it allows for the unification of various attributes across different pattern selectors. However, it can potentially lead to unexpected unification of logical variables when composing *named* patterns. For example, our first implementation of scenario 5 in figure 4.9 contains a bug as it incorrectly does not match the sequence of messages shown in figure 4.10.A. Figure 4.10.B shows the expanded form of the composite pattern defined in figure 4.9. Each elementary pattern's selector contains the same logical variable `id`. However, in this case, it is undesirable to unify these three logical variables as each sensor can have a different id. To circumvent this issue, Sparrow allows developers to manually change the logical variable for an attribute using the aliasing operator (~>). The aliasing operator renames the logical variable on its left-hand side to the logical variable on its right-hand side in the elementary pattern.

We acknowledge this operator is not an optimal solution. In future versions of Sparrow, the default behaviour of unification may be changed to facilitate the maintenance of large pattern sets. However, as patterns are not first-class entities, developers can identify shared logic variables by looking at the pattern definition.

Figure 4.10.C presents an updated version of the `occupied_home` pattern defined in figure 4.9. The updated definition of the `occupied_home` pattern reflects two changes. First (C.1), the `id` attribute of the second pattern (`:contact`) was manually renamed to `cid`. Second (C.2), the `id` attribute of the third pattern (motion - entrance hall) was renamed using the *alias* (`~>`) operator.

4.3.2.4 Timing Constraints on Composite Patterns

Similar to windowing for elementary patterns, composite patterns also support *timing constraints*. Developers can specify a *time-interval* in which the composed set of constituent patterns should be matched. Figure 4.11 presents a new version of the `occupied_home` pattern where the 60 seconds time constraint is added. Similar to the intentional sequencing operator (`:seq`), this time constraint (`interval`) is also defined in the options of a composite pattern.

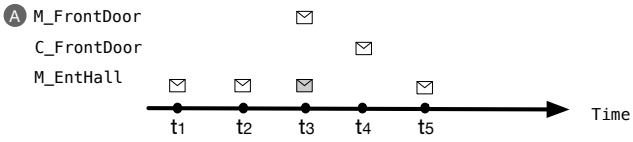
```
pattern occupied_home as motion_sensor{location= :front_door}
                      and {:contact, cid, :open, :front_door}
                      and motion_sensor{location= :entrance_hall,id~> mid},
                      options: [ seq: true, interval: {60, :secs} ]
```

Figure 4.11 Example of a composite pattern with a time interval constraint

4.3.2.5 Composite Patterns Selection Strategy

Like most actor languages, Sparrow messages are matched in FIFO[5] order. However, Sparrow allows developers to deviate from that default selection strategy. Figure 4.5 already showed an example of this action by only selecting every third message. A flexible message selection strategy is also useful to synchronize always on the latest messages that may be relevant to

[5]FIFO - **F**irst **I**n **F**irst **O**ut




```
B pattern occupied_home as motion_sensor{location= :front_door}
                        and {:contact, id, :open, :front_door}
                        and motion_sensor{location= :entrance_hall, id ~> mid},
                        options: [ seq: true, interval: {60, :secs},last: true ]
```

■ **Figure 4.12** A solution to the occupied-home scene of scenario 5: (A) Messages received by the actor; (B) Composite pattern that enforces a selection strategy (*last-in*)

a pattern. For example, the implementation of the pattern `occupied_home` from figure 4.11 must always check all potential messages received from the three sensors in the last 60 seconds. However, as observed in figure 4.12.A at t_4, the pattern should only check the latest message (t_3) from the entrance hall's motion sensor and discard the old ones (t_1, t_2).

4.3.3 Accumulation Patterns

The third big category of patterns (often elementary and composite) are accumulation patterns. Accumulation patterns extend the above patterns with *quantified* and *unquantified* accumulation of messages. Once the messages are accumulated, they can be subject to further transformation and filtering operations. Accumulation patterns can be constructed by means of several optional accumulation operators. Like other operators (see section 4.3.1.1), they can be specified between square brackets following the selector.

4.3.3.1 Quantified Accumulation

Quantified accumulation patterns are used to accumulate a certain number of matching messages. Figure 4.13.A shows a pattern that uses the `count` operator to match three heating failure messages. Like composite patterns, quantified accumulation patterns do a unification of their logic variables during the expansion of its pattern selector. In our example, this means that the pattern `heating_failure` accumulates three heating failure messages

```
A pattern heating_failure as {:heating_f, id, code}[count: 3]

B {:heating_f, id, code} and {:heating_f, id, code} and {:heating_f, id, code}

C pattern heating_failure as {:heating_f, id, @code}[count: 3]
```

Figure 4.13 Example of a quantified accumulation pattern that matches three heating failure messages

from the same boiler and with the same failure code as the logical variables `id` and `code` will be unified (see figure 4.13.B).

As we mentioned in section 4.3.2.3, the above default behaviour may be useful in some circumstances, but it is not always desirable. Sparrow introduces the operators: *must be distinct* "`!`" and *may be distinct* "`@`", to specify the expected matching behaviour of logic variables when used in an accumulation pattern. The former guarantees that the constrained attribute must have a *distinct* value in *all* the messages accumulated. In contrast, the latter allows the constrained attribute to have *any* value. For example, the definition of `heating_failure` pattern shown in figure 4.13.C will match three messages from the same boiler regardless of the `code` attribute's value.

4.3.3.2 Unquantified Accumulation

Unquantified accumulation patterns are used to accumulate any number of messages within a certain time window using the `window` operator. Figure 4.14.A shows a pattern that uses the `window` operator to accumulates all heating failure messages in the last 60 minutes. Messages older than the time constraint are automatically removed by Sparrow's run-time. As a second example of unquantified accumulation patterns, figure 4.14.B shows a hybrid accumulation pattern that use both `count` and `window` accumulation operators. In this last case, the first operator that reaches its condition will win and result in a match.

4.3.3.3 Transformation operators

Transformation operators allow developers to transform and filter a group of messages that have been accumulated using the previously described operators. Inspired by functional languages such as Haskell and Elixir,

```
A pattern heating_failure as {:heating_f, id, @code}[window: {60, :mins}]

B pattern heating_failure as {:heating_f, id, @code}[count: 3, window: {60, :mins}]
```

Figure 4.14 Examples of unquantified accumulation patterns: (A) Use of the `window` operator to accumulate all messages in the last 60 minutes; (B) Example mixing both accumulation operators

Sparrow supports two transformation operators *map* and *fold*. The behaviour of these operators resembles the ones from the same abstractions found in the above mentioned functional languages.

Figure 4.15.B presents a `electricity_alert` pattern that implements the requirements for scenario 6. For this scenario, it is required to accumulate the daily electricity consumption for the last three weeks and check if the total consumption was greater than 200 kWh. Notice in our implementation, the use of the *may be distinct* operator (B.1) to match any daily consumption values. Once the accumulation operator of pattern `electricity_alert` is satisfied (B.2), the list of messages is transformed (B.3). Later, the pattern uses the special form `bind` (B.4) to bind the total electricity consumption in a local variable (*total*), which scope extends to the rest of the pattern. Finally, the guard expression (B.5) is evaluated to determine if the messages are consumed or not by the pattern. In this example, messages older than three weeks are automatically discarded by Sparrow's runtime.

```
A pattern daily_electricity as {:consumption, meter_id, value}

                                                  1          2
B pattern electricity_alert as daily_electricity{@value}[window: {3, :weeks}]
                               |> fold(0, fn(_,_,v}, acc)-> acc + v end)     3
                               |> bind(total)                               4
                               when total > 200                             5
```

Figure 4.15 Sparrow solution for scenario 6: (A) Elementary pattern definition; (B) Accumulation pattern example with a transformer operator and guard

In this section, we have focused on the definition of patterns to compose the interface of an actor. However, we have not specified what should be

done with the matched messages. The next section will introduce a new abstraction called *reaction*, which is used to define the logic to process the matched messages of a pattern.

4.4 Sparrow's Reaction Language

In Sparrow, once a pattern is matched, the matched messages are consumed, i.e., removed from the actor's inbox, and the actor starts reacting to the matched pattern. A reaction is a piece of code that corresponds to a "function" in Elixir (see figure 4.16). Sparrow's reactions always receive the three parameters. The first one (A.1) is a list of messages matched by the pattern. The second parameter (A.2) is a key-value list with all the intermediate results saved with the `bind` operator during a transformation process (see figure 4.15.B.4). Finally, the third parameter (A.3) represents the current state of the actor.

In Sparrow, this reaction logic is syntactically decoupled from the pattern definition (similar to [83]). A reaction can be dynamically bound to one or more patterns, and a pattern can ever have multiple reactions. This behaviour was intentionally designed to facilitate the reuse of both patterns and reactions. Furthermore, it avoids developers adding an extra layer of indirection (middleman) to determine which reactions to execute. The middleman approach circumvents the duplicity definition of both patterns and reactions. However, it also introduces a performance penalty since it is always notified whether a pattern has registered reactions or not. In brief, the decoupling of patterns and reactions allows the programmer to change the behaviour of an actor dynamically, which is reminiscent of a *become* statement in the original actor model.

Binding a reaction to a pattern is done using the `react_to` special form (see figure 4.16.C) which expects two arguments: the *pattern's name* (C.1), and the *reaction's name* (C.2). When the pattern `open_window` (see figure 4.16.B) is successfully matched, the actor will invoke all its reactions in the same order they were bound to the pattern. A reaction can also be unbound from a pattern. The `remove` primitive function (see figure 4.16.D) also expects the same two arguments as the `react_to` primitive function but in reverse order. Finally, the `remove_reactions` function (see figure 4.16.E) removes all the reactions of the pattern received as argument.

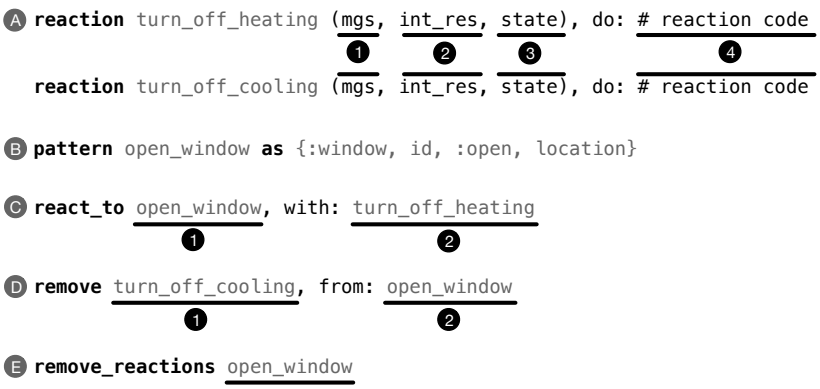


Figure 4.16 Overview of reaction primitives

Figure 4.16 showcases a smart home scenario where the binding of reactions to a pattern is based on the current season of the year (e.g., summer, winter). In this way, developers can avoid duplicated patterns with different reactions to match the season's requirements. For clarity, this figure omits the code related to the dynamic scheduling of the reactions. Figure 4.16.A shows the definition of two reactions (`turn_off_heating`, `turn_off_cooling`) using the `reaction` special form. Both reactions can be bound to the pattern `window_open` (see figure 4.16.B) based on the current season.

4.5 Conclusion

By implementing the synchronizations requirements identified in section 2.3 as language constructs, Sparrow has moved forward to define complex coordination patterns easily in an actor-based language. Although we have demonstrated the need for such language constructs, other application domains might need new synchronization abstractions. Sparrow's host language provides an advanced macro system to extend its language capabilities. However, it also restricts the expressiveness of its domain-specific languages to a set of predefined operators supported by its parser. The unification of logic variables of a single pattern's selector contributes to declare more expressive patterns. Still, further research is needed to confirm the relevance of its application across multiple selectors of a composite pattern.

CHAPTER 5

NEST: A Formal Semantics of Sparrow

In the previous chapter, we introduced Sparrow as a domain-specific language for orchestrating actors which e.g., represent physical objects in cyber-physical systems. We detailed the main abstractions of its internal pattern language (see section 4.3) and reaction language (see section 4.4), but not their formal operational semantics. This chapter presents a formal calculus of Sparrow called NEST[1]. We use this formal calculus to precisely describe the semantics of core coordination abstractions presented in chapter 4. First, we describe NEST's syntax and small-step operational semantics (see section 5.1). Second, we introduce an implementation of NEST in Redex (see section 5.2). We use this executable NEST implementation to validate its formal semantics definition by means of an extensive suite of hand-written and randomized tests. Section 5.2.2 describes how we extended the randomized test suite provided by Redex to evaluate NEST pattern-matching semantics' correctness. We end this chapter by listing the differences between the formal semantics of NEST and the current implementation of Sparrow (see section 5.2.3).

[1]NEST: NEST Epitomises Sparrow Theory

$$
\begin{array}{rcl}
P_s \subseteq \textbf{Pattern} & ::= & \texttt{pattern}\ p_n\ \texttt{as}\ p \\
R_s \subseteq \textbf{Reaction} & ::= & \texttt{reaction}\ r_n\ \texttt{do}\ e \\
A_s \subseteq \textbf{Actor} & ::= & \texttt{actor}\ a_n\ \{\overline{\texttt{react\_to}\ p_n\ \texttt{with:}\ r_n}\} \\
e \in E \subseteq \textbf{Expr} & ::= & nil \mid x \mid l \mid i \mid \lambda\overline{x}.e \mid \{m_{id}, \overline{e}\} \mid \texttt{let}\ x = e\ \text{in}\ e \mid \texttt{spawn}\ a_n \\
 & & \mid \texttt{send}\ e, e \mid \texttt{react\_to}\ p_n, \texttt{with:}\ r_n \mid \texttt{remove}\ r_n, \texttt{from:}\ p_n \\
 & & \mid \texttt{remove\_reactions}\ p_n
\end{array}
$$

$x, l, i \in \textbf{VarName}$, $p_n \in \textbf{PatternName}$, $r_n \in \textbf{ReactionName}$, $a_n \in \textbf{ActorName}$

Figure 5.1 Abstract syntax of NEST

5.1 Operational Semantics

This section starts by defining the syntax of NEST (see section 5.1.1) and its main semantic entities (section 5.1.2). Next, it introduces a set of reduction rules based on evaluation contexts [21] to describe the core features supported by NEST (section 5.1.3).

5.1.1 Syntax

The NEST abstract syntax is composed of three main subsets, whose basic abstractions can be observed in figure 5.1.

Pattern subset (P_s) describes all different types of patterns introduced in section 4.3. In this abstract definition, p_n represents the name of the pattern, and p is used as a placeholder for the non-terminal expression `<pattern>` of figure 5.2 previously defined in chapter 4.

Reaction subset (R_s) describes the definition of reaction functions. A reaction function has a name r_n and a body e. The body e of a reaction in NEST has access to three implicit arguments i.e., *list of matched messages*, *list of intermediate results*, and the *current state of the actor*. In case, the *list of intermediate results* correspond to transformation executed over the accumulation patterns defined in section 4.3.3. We opted for this variant of implicit arguments to simplify the implementation of the operational semantics.

Actor subset (A_s) describes how to define new actor behaviours. The `actor` construct takes two arguments: the actor name a_n, and a set of bindings between patterns and reactions previously defined.

We integrate the above subsets into NEST's expressions e together with a set of built-in functions to create NEST's programs. For example, the built-in function `react_to with:` binds a pattern with name p_n to a reaction with name r_n. As patterns and reactions are second-class entities, their lookup and binding (association), is globally scoped within an actor. The function `remove from:` removes a reaction with name r_n from a pattern with name p_n. Similarly, `remove_reactions` removes all reactions associated to a pattern with name p_n. After the definition of an actor, several actors can be created using the `spawn` primitive. This function takes a single argument, the actor name a_n, and returns the id (a_{id}) of the newly spawned actor. Finally, lambdas are denoted by $\lambda\overline{x}.e$, and local variables can be introduce via `let` $x = e$ in e.

5.1.2 Semantic Entities

Figure 5.3 shows the primary semantic entities of a NEST program. Here, we use different font styles to distinguish our semantic entities syntactically. For example, the calligraphic letter $\mathcal{A}$ represents a constructor. Regular uppercase letters like K and M denote sets or sequences. Parentheses represent a pair, while curly brackets denote a tuple as in Elixir. Actors and messages have different types of identifiers (or ids for short), denoted a_{id} and m_{id} respectively.

The computational state of a NEST program is represented as a configuration K. This configuration is composed of a set of running actors. Each actor consists of an identifier a_{id}, a queue Q of messages remaining to be processed (a.k.a. the *inbox*), a set of pattern-reaction bindings $\overline{(p_n, r_n)}$, and an expression e that is currently evaluated. NEST and Sparrow inherit all primitive values supported by Elixir. To simplify the semantics of NEST, only primitive values used by its reduction rules are included (e.g., `nil`, a_{id}).

Similar to Elixir, messages are represented as tuples, where the first element m_{id} denotes the type of a message followed by a list of values $\overline{v}$. Intermediate results of pattern transformations I are represented as a set of values v or pairs $\overline{x, v}$. The first element of a pair corresponds to a variable's

```
<p-definition>      :=  pattern <identifier> as <pattern>
<r-definition>      :=  reaction <identifier> (<arg>, <arg>, <arg>) do <expression> end
<react-to>          :=  react_to <identifier>, with: <identifier>
<remove-from>       :=  remove <identifier>, from: <identifier>
<remove-reactions>  :=  remove_reactions <identifier>
<pattern>           :=  <comp-pattern> [<guard>] [, [options: <option>+]]
<comp-pattern>      :=  <elem-pattern> [(and | or ) <elem-pattern>]*
<elem-pattern>      :=  [not] <selector> [[<operator>+]] {|> <transformer>}*
<selector>          :=  {<symbol>, <attribute>*}
                        | <identifier>[{{<inline-guard>
                        | <alias-op>}+}]
<attribute>         :=  <value>
                        | <symbol>
                        | <logic-var>
<guard>             :=  when <expression>
<inline-guard>      :=  <identifier> = <expression>
<alias-op>          :=  <identifier> ~> <identifier>
<symbol>            :=  :<identifier>
<logic-var>         :=  [(@ | ! )]<identifier>
<operator>          :=  window: <time>
                        | debounce: <time>
                        | every: <number>
                        | count: <number>
<transformer>       :=  fold(<expression>, <expression>)
                        | bind(<identifier>)
<option>            :=  seq: <boolean>
                        | interval: <time>
                        | last: <boolean>
<time>              :=  {<number>, (:secs | :mins | :hours | :days | :weeks)}
<arg>               :=  <identifier>
```

Figure 5.2 Sparrow EBNF-styled syntax (repetition of figure 4.1)

name and the second to its value. These variables can be used in guard expressions, and they are also accessible in the body of a reaction. Finally, the subset of expressions e is extended to include actor identifiers a_{id} since a sub-expression may reduce to an actor identifier before being reduced further. This new subset of expressions is called *runtime expressions*.

5.1.3 Reduction Rules

This section describes NEST's reduction rules. These rules are based on evaluation contexts [21], and they operate on the *runtime expressions* defined in section 5.1.2.

$$
\begin{array}{rcll}
K \in \textbf{Configuration} & ::= & A & \text{Configurations} \\
a \in A \subseteq \textbf{Actor} & ::= & \mathcal{A}\langle a_{id}, Q, \overline{(p_n, r_n)}, e\rangle & \text{Actors} \\
M, L \in \textbf{MessageList} & ::= & \overline{\{m_{id}, \overline{v}\}} & \text{List of Messages} \\
v \in \textbf{Value} & ::= & \texttt{nil} \mid a_{id} & \text{Values} \\
I \subseteq \textbf{IntResult} & ::= & \overline{v \mid (x, v)} & \text{Intermediate Results} \\
e \in E \subseteq \textbf{Expr} & ::= & \ldots \mid a_{id} & \text{Runtime Expressions}
\end{array}
$$

$$a_{id} \in \textbf{ActorId},\ m_{id} \in \textbf{MessageId},\ x \in \textbf{VariableName},$$

Figure 5.3 Semantic entities of NEST

5.1.3.1 Evaluation Contexts

An evaluation context is a term that contains a hole $e_\square$. A hole has the purpose of identifying the next subexpression to reduce in a compound expression. NEST reduction rules use evaluation contexts to determine which subexpressions should be entirely reduced to a value before the compound expression itself can be reduced. In other words, the reduction rule will try to find the sub-term that matches the hole. We use the notation $e_\square[e]$ to indicate that the expression e is part of a compound expression $e_\square$. Consequently, a set of reduction rules should reduce such expression e before they reduce the compound expression $e_\square$ further.

$$e_\square \quad ::= \quad \square \mid \texttt{let}\ x\ =\ e_\square\ \texttt{in}\ e \mid \{m_{id}, \overline{v}, e_\square, \overline{e}\} \mid \texttt{send}(e_\square, e) \mid \texttt{send}(e, e_\square)$$

In NEST, holes can appear in two main expressions: `let` and `send`. A `let` expression can only be further reduced once the left expression is fully reduced to a value. Expressions that are part of a message tuple are always reduced from left to right. Similarly, the `send` primitive reduces its left argument before its right argument.

5.1.3.2 Notation

We use the following conventions to facilitate the understanding of NEST evaluations rules. The notation $O = O' \cdot O''$ concatenates two sequences or lists. Sometimes, we also use $O = O' \cdot \mathrm{o}$ as a shortcut to concatenate two lists. In this particular case o is considered as a list with a single element.

$$
\begin{array}{rcl@{\qquad}rcl}
[v/x]x' & = & x' & [v/x]\texttt{let } x' = e \texttt{ in } e & = & \texttt{let } x'[v/x]e \text{ in } [v/x]e \\
[v/x]x & = & v & [v/x]\texttt{let } x = e \texttt{ in } e & = & \texttt{let } x = [v/x]e \texttt{ in } e \\
[v/x]\text{nil} & = & \text{nil} & [v/x]a_{id} & = & a_{id}
\end{array}
$$

$$
\begin{array}{rcl}
[v/x]\texttt{send } e, e & = & \texttt{send } [v/x]e,\ [v/x]e \\
[v/x]\texttt{spawn } p_n & = & \texttt{spawn } p_n \\
[v/x]\texttt{react\_to } p_n,\ r_n & = & \texttt{react\_to } p_n,\ r_n \\
[v/x]\texttt{remove } r_n,\ \texttt{from:} p_n & = & \texttt{remove } r_n,\ \texttt{from:} p_n \\
[v/x]\texttt{remove\_reactions } p_n & = & \texttt{remove\_reactions } p_n
\end{array}
$$

■ **Figure 5.4** Substitution rules: x denotes a variable name or the pseudo-variable, v denotes a value.

Simultaneously, we use a similar notation to deconstructs a sequence being part of the arguments of an auxiliary function. For example, we use the notation $\text{o} \cdot O$ and $O \cdot \text{o}$ to remove the first or last element respectively from the sequence O. We represent queues as sequences of messages that are processed right-to-left, meaning that the last message in the sequence is the first one to be processed. We use the $\emptyset$ symbol to denote both empty sets and empty sequences.

5.1.3.3 Evaluation Rules

NEST's semantics is defined in terms of a relation $\rightarrow$ on configurations, $K \rightarrow K'$. A NEST program is an expression e that is reduced to an initial configuration containing a single "root" actor $K_{init} = \{\mathcal{A}\langle a'_{id}, \emptyset, \emptyset, \texttt{nil}\rangle\}$. The actor's queue and the pattern-reaction binding list are initially empty. NEST does not yet consider actors distributed across different devices; it relies on the default behaviour of the base actor language to handle disconnections. In other words, NEST assumes that all actors remain permanently connected.

The evaluations rules described in this section are split into two groups to make explicit which actions can be executed in isolation by a single actor (*local-rules*), and which ones require interaction between different actors of a configuration (global-rules). The above rules can be applied non-deterministically, which gives rise to concurrency. Figure 5.4 outlines the set of substitution rules employed by NEST evaluation rules.

Actor-local reduction rules are always initiated after taking the next message from their actor's message queue. Such a message is then trans-

$$\frac{}{\begin{array}{c}\mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}, e_\square[\text{let } x = v \text{ in } e]\rangle \uplus A \\ \rightarrow \mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}, e_\square[[v/x]e]\rangle \cup A\end{array}} \quad \text{(LET)}$$

$$\frac{M', L, I, e = \texttt{match}(\overline{(p_n, r_n)}, M)}{\begin{array}{c}\mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}, v\rangle \uplus A \\ \rightarrow \mathcal{A}\langle a_{id}, M', \overline{(p_n, r_n)}, [L/l][I/i]e\rangle \cup A\end{array}} \quad \text{(MATCH-MESSAGE)}$$

$$\frac{}{\begin{array}{c}\mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}, e_\square[\texttt{react\_to}(p_n, r_n)]\rangle \uplus A \\ \rightarrow \mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)} \cdot (p_n, r_n), e_\square[nil]\rangle \cup A\end{array}} \quad \text{(REACT-TO)}$$

$$\frac{\overline{(p_n, r_n)}' = \texttt{remove}(r_n, p_n, \overline{(p_n, r_n)})}{\begin{array}{c}\mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}, e_\square[\texttt{remove}(r_n, p_n)]\rangle \uplus A \\ \rightarrow \mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}', e_\square[nil]\rangle \cup A\end{array}} \quad \text{(REMOVE-REACTION-FROM)}$$

$$\frac{\overline{(p_n, r_n)}' = \texttt{remove\_reactions}(p_n, \overline{(p_n, r_n)})}{\begin{array}{c}\mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}, e_\square[\texttt{remove\_reactions}(p_n)]\rangle \uplus A \\ \rightarrow \mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}', e_\square[nil]\rangle \cup A\end{array}} \quad \text{(REMOVE-REACTIONS)}$$

Figure 5.5 Actor-local reduction rules.

formed into an expression (e.g., by a reaction's body) that is finally reduced to a value. The process of reducing such a single expression to a value is called a *turn*. Once a turn is completed, the next message is processed. The activation of local rules stops when the actor's message queue is empty, and it restarts when a new message arrives. If during the activation of a local rule, it cannot reduce an expression further, the actor is said to be stuck. This unexpected behaviour signifies a semantic error in the program. Figure 5.5 lists the set of actor-local reduction rules implemented by NEST.

- LET: a "let" - expression simply substitutes the value of x for v in e according to the substitution rules outlined in Figure 5.4.
- MATCH-MESSAGE: this rule matches messages in the actor's inbox with that actor's interface. Its activation only occurs when the expression of the actor's previous turn is fully reduced to a value.

The auxiliary function `match` is then used to match the actor's pattern-reaction pairs `(pr, rn)` with that actor's inbox, M. When a match is found, this auxiliary function returns the updated inbox, M', without the matched messages, the list of matched messages, L, a list of bound identifiers, I, and the expression, e, of the reaction that was associated with the matched pattern. The actor's inbox is updated, and the currently active expression is substituted with the reaction body, e. In this expression, the variables l and i are replaced with the list of matched messages and the bound identifiers respectively.

- REACT-TO: this rule associates a pattern named p_n with a reaction named r_n. The new association is added to the right of the list of pattern-reaction bindings. The `react-to` expression is then reduced to `nil`. Unlike in Sparrow, NEST patterns and reaction are globally defined. This decision helped us to simplify the formal description of our model.

- REMOVE-REACTION-FROM: this rule removes the association between a reaction named r_n and a pattern named p_n. The auxiliary function `remove` (see figure 5.8) removes the particular entry from the pattern-reaction list associated with the specified reaction and pattern. Furthermore, it returns the updated set of pattern-reaction bindings. The `remove-reaction-from` expression is then reduced to `nil`.

- REMOVE-REACTIONS: this rule removes all associations of reactions from a pattern p_n, where p_n the name of the pattern. The auxiliary function `remove_reactions` (see figure 5.8) removes the entries from the pattern-reaction list associated with the specified pattern. Furthermore, it returns the updated set of pattern-reaction bindings. The `remove-reactions` expression is then reduced to `nil`.

Actor-global reduction rules formalize operations that require interaction between different actors of a configuration. Figure 5.6 shows the two actor-global reduction rules supported by NEST.

- SPAWN-ACTOR: this rule describes the reduction of spawning a new actor. When the `spawn-actor` expression is reduced, a new actor with identifier a'_{id} is added to the configuration. Furthermore, it

$$\frac{\begin{array}{c} a'_{id} \ \texttt{fresh} \\ \texttt{actor}\ a_n\{\overline{(p_n, r_n)}'\} \in A_c \end{array}}{\begin{array}{c} \mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}, e_\square[\texttt{spawn}(a_n)]\rangle \cup A \\ \rightarrow \mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}, e_\square[a'_{id}]\rangle \cup \mathcal{A}\langle a'_{id}, \{\texttt{:init}\}, \overline{(p_n, r_n)}', \texttt{nil}\rangle \cup A \end{array}} \quad \text{(SPAWN-ACTOR)}$$

$$\frac{}{\begin{array}{c} \mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}, e_\square[\texttt{send}(a'_{id}, \{m_{id}, \overline{v}\})]\rangle \cup \\ \mathcal{A}\langle a'_{id}, M', \overline{(p_n, r_n)}', e'\rangle \cup A \\ \rightarrow \mathcal{A}\langle a_{id}, M, \overline{(p_n, r_n)}, e_\square[nil]\rangle \cup \mathcal{A}\langle a'_{id}, \{m_{id}, \overline{v}\}.M', \overline{(p_n, r_n)}', e'\rangle \cup A \end{array}} \quad \text{(SEND-MESSAGE)}$$

Figure 5.6 Actor-global reduction rules.

returns the identifier to the new actor, allowing the creator actor to communicate further with the new actor. The inbox of the new actor is populated with an `{:init}` message that will use as a starting point to initialize the actor's interface. The interface of an actor is defined by the set of messages or message patterns that the actor understands. In NEST, an actor's interface is defined by that actor's pattern-reaction pairs.

- SEND-MESSAGE: this rule describes the reduction of an asynchronous message sent to an actor. After a message is sent, a new message is added to the inbox of the recipient actor a'_{id}, and the expression is then reduced to *nil*. NEST only formalizes one-way asynchronous messages as in Elixir.

Figures 5.7 and 5.8 list the most important auxiliary functions used by NEST's reduction rules. Some of the less relevant auxiliary functions are not listed in the interest of clarity, but we describe their purposes below. The interested reader can find the implementation of all auxiliary functions in our GitHub repository[2].

- `match`: this auxiliary function matches the entire inbox of an actor with that actor's pattern reaction pairs. When it finds a set of messages that satisfy the conditions of a particular pattern, it removes them from the actor's inbox. Then it returns the updated inbox M' together with the list of matched messages L, the list of bound identifiers I and the body expression e of the associated reaction.

[2]NEST Implementation - `https://github.com/rhumbertgz/nest-plt-redex`

- `match_patt`: this overloaded auxiliary function formalises the core of the Sparrow pattern language. It returns an updated inbox, a list of matched messages, a list of bound identifiers and a list of bindings between logical variables and primitive values. The latter list is used for unification when finding a match for a compound pattern. We added a numeric label to each of its variants to facilitate their description (see figures 5.7 and 5.8).

    1. Match a pattern composed by a selector against the messages in the actor's inbox. In this case, relevant messages are determined by pattern-matching their attribute values against the pattern's selector.
    2. Match a pattern composed by a conjunction between the patterns `p1` and `p2` against the messages in the actor's inbox. In this case, the function `unify` is used to check that shared logic variables between `p1` and `p2` have the same value. The matching order of messages is not relevant.
    3. Match a pattern composed by a pattern `p1` followed by a pattern `p2` against the messages in the actor's inbox. The function `unify` is used to check that shared logic variables between `p1` and `p2` have the same value. Additionally, this match function checks that a relevant message for `p2` arrived to the actor's inbox after a relevant message for `p1`.
    4. Match a pattern composed by a disjunction between `p1` and `p2` against the messages in the actor's inbox. In this case, a match will occur whenever a relevant message for `p1` or `p2` is found.
    5. Match a pattern `p` against the messages in the actor's inbox. In this case, only messages that satisfy the guard ($\xrightarrow{\text{h}}$ *) expression e are considered relevant.
    6. Match an accumulation pattern `p` with the operator `count` against the messages in the actor's inbox. A match will occur when `n` relevant messages are found. In this case, a list of `n` messages will be returned.
    7. Match a pattern p with the operator `every` against the messages in the actor's inbox. A match will occur when `n` relevant messages are found. In this case, it only returns a list with the last (n) message matched.

8. Match an accumulation pattern p with the operator `window` against the messages in the actor's inbox. A match will occur when any relevant messages are found within the specified time window. In this case, a list of all matched messages will be returned.
9. Match an extension of the pattern p_n against the messages in the actor's inbox. Remember that we mentioned in section 4.3.2.1 that Sparrow's can be reused and extended.
10. Match an accumulation pattern `p` with the transformation operator `fold` against the messages in the actor's inbox. After the conditions of `p` are satisfied, the `fold` operator is applied to the list of matched messages.
11. Match an accumulation pattern `p` with the transformation operator `map` against the messages in the actor's inbox. After the conditions of `p` are satisfied, the `map` operator is applied to the list of matched messages.
12. Match an accumulation pattern `p` with the operator `bind` against the messages in the actor's inbox. After the conditions of `p` are satisfied, the bind operator is applied to save intermediate results after a transformation operator is applied.

- `match_attr`: this auxiliary function matches a pattern's attributes with a message's values. A value attribute in the pattern must always be identical to each corresponding value in the message. A logical variable always matches with any value in the message. This function returns a list of bindings between these logical variables and their corresponding values in the message.

- `within_outside_window`: this auxiliary function determines if potential messages for a pattern's match are within or outside a particular time window. It takes a time interval, `{n, unit_time}`, and an inbox, M, and returns two separate inboxes, M' and M''. The former is a list of messages that fall within the window. The latter is a list of messages that fall outside of the window.

- `seq?`: this auxiliary function determines if the first message from the list L' arrived at the actor's inbox after the last message of list L.

$$
\begin{array}{rcl}
\texttt{match}((p_n, r_n) \cdot \overline{(p_n, r_n)},\ M) & = & M', L, I, e \\
 & & \textit{where} \\
 & & \texttt{pattern}\ p_n\ \texttt{as}\ p \in P_L \\
 & & \texttt{reaction}\ r_n\ \texttt{do}\ e \in R_L \\
 & & M', L, I, (lv, v) = \texttt{match\_patt}(p, M) \\
 & & \\
\texttt{match}((p_n, r_n) \cdot \overline{(p_n, r_n)},\ M) & = & \texttt{match}(\overline{(p_n, r_n)},\ M) \\
 & & \\
(1)\ \texttt{match\_patt}(\{m_{id}, \overline{at}\},\ M \cdot \{m_{id}, \overline{v}\}) & = & M, \{m_{id}, \overline{v}\}, \emptyset, \overline{(lv, v)} \\
 & & \textit{where} \\
 & & \overline{(lv, v)} = \texttt{match\_attr}(\overline{at}, \overline{v}) \\
 & & \\
(2)\ \texttt{match\_patt}(p_1\ \texttt{and}\ p_2, M) & = & M'', L \cdot L', I \cdot I', \texttt{unify}(\overline{(lv, v)}, \overline{(lv', v')}) \\
 & & \textit{where} \\
 & & M', L, I, \overline{(lv, v)} = \texttt{match\_patt}(p_1,\ M) \\
 & & M'', L', I', \overline{(lv', v')} = \texttt{match\_patt}(p_2,\ M') \\
 & & \\
(3)\ \texttt{match\_patt}(p_1\ \texttt{andThen}\ p_2, M) & = & M'', L \cdot L', I \cdot I', \texttt{unify}(\overline{(lv, v)}, \overline{(lv', v')}) \\
 & & \textit{where} \\
 & & M', L, I, \overline{(lv, v)} = \texttt{match\_patt}(p_1,\ M) \\
 & & M'', L', I', \overline{(lv', v')} = \texttt{match\_patt}(p_2,\ M') \\
 & & \textit{seq?}(L, L') \\
 & & \\
(4)\ \texttt{match\_patt}(p_1\ \texttt{or}\ p_2,\ M) & = & \texttt{match\_patt}(p_1,\ M) \\
\texttt{match\_patt}(p_1\ \texttt{or}\ p_2,\ M) & = & \texttt{match\_patt}(p_2,\ M) \\
 & & \\
(5)\ \texttt{match\_patt}(p\ \texttt{when}\ e,\ M) & = & M', L, I, \overline{(lv, v)} \\
 & & \textit{where} \\
 & & M', L, I, \overline{(lv, v)} = \texttt{match\_patt}(p,\ M) \\
 & & \overline{[v/lv]}e \xrightarrow{\text{h}}{}^{*} \text{true} \\
 & & \\
\texttt{match\_patt}(p\ \texttt{[count: 0]},\ M) & = & M', \emptyset, \emptyset, \emptyset \\
(6)\ \texttt{match\_patt}(p\ \texttt{[count: } n\texttt{]},\ M) & = & M'', L \cdot L', I \cdot I', \texttt{unify}(\overline{(lv, v)}, \overline{(lv, \overline{v'})}) \\
 & & \textit{where} \\
 & & M', L, I, \overline{(lv, v)} = \texttt{match\_patt}(p,\ M) \\
 & & M'', L', I', \overline{(lv, \overline{v'})} = \texttt{match\_patt}(p,\ \texttt{[count: } n-1\texttt{]},\ M') \\
 & & \\
\texttt{match\_patt}(p\ \texttt{[every: 1]},\ M) & = & \texttt{match\_patt}(p,\ M) \\
(7)\ \texttt{match\_patt}(p\ \texttt{[every: } n\texttt{]},\ M) & = & \texttt{match\_patt}(p,\ \texttt{[every: } n-1\texttt{]},\ M') \\
 & & \textit{where} \\
 & & M', L, I, \overline{(lv, v)} = \texttt{match\_patt}(p,\ M) \\
 & & \\
(8)\ \texttt{match\_patt}(p\ \texttt{[window: } \{n, ut\}\texttt{]},\ M) & = & M'' \cdot M''', L, I, \overline{(lv, \overline{v})} \\
 & & \textit{where} \\
 & & M', M'' = \texttt{within\_outside\_window}(\{n, ut\},\ M) \\
 & & M''', L, I, \overline{(lv, \overline{v})} = \texttt{match\_patt}(p,\ M')
\end{array}
$$

Figure 5.7 Auxiliary functions used in the reduction rules.

It is assumed that both lists have their elements ordered by their arrival time to the actor's inbox.

- `expr` $\xrightarrow{\texttt{h}}{}^{*}$ `boolean`: defines a reduction relation $\xrightarrow{\texttt{h}}{}^{*}$ that reduces a predicate expression (`expr`) in the host language (`h`) to a boolean value. We do not specify this relation here. However, we apply it here repeatedly to fully reduce the guard expression to a boolean value. If the constraints of the guard are satisfied, the expression will reduce to true.

- `foldl`: this auxiliary function resembles a standard `foldl` function available on functional programming languages. It applies the λ function for each element in the list L with the neutral element n.

- `map`: this auxiliary function resembles a standard `map` function available on functional programming languages. It applies the λ function for each element in the list L.

- `fresh` creates a new variable identifier.

5.2 NEST Calculus in Redex

Previous sections of this chapter have focused on NEST's formal model for Sparrow. Although we consider that the above model manages to describe the main language design ideas of our DSL, we did not state any theorems to prove its "correctness". Since a faulty model does not serve its purpose, we opted to implement a mechanized model of NEST in order to prove its correctness. However, instead of stating theorems and creating heavy-weight machine-checked proofs (e.g., in Coq [71]), we chose a lightweight approach based on random testing. We based our decision inspired by the results showed in [41], where the authors demonstrated the effectiveness of random testing by detecting mistakes in nine formalized models, including two, which were already mechanized. Furthermore, we wanted a lightweight mechanism that did not required more explicit details than programming.

In this section, we introduce **a mechanized implementation of NEST** in Redex [21]. Redex is a domain-specific language in Racket [78] for formalizing operational semantics. It allowed us as language designers to write down NEST language grammar, reduction rules, and auxiliary functions in an untyped and high-level expressive language. Furthermore, its integrated set of tools (e.g., unit-test suite, visual inspector) helped

$$
\begin{array}{rcl}
\text{(9) } \texttt{match\_patt}(p_n,\ M) & = & \texttt{match\_patt}(p',\ M) \\
 & & \textit{where} \\
 & & \texttt{pattern}\ p_n\ \texttt{as}\ p \in P_L \\
 & & \overline{lv} = \mathit{logical\_vars}(p) \\
 & & p' = \overline{[lv'/lv]}p \qquad\qquad \overline{lv'}\ \texttt{fresh} \\
 & & \\
\text{(10) } \texttt{match\_patt}(p\ \texttt{fold}(n, \lambda),\ M) & = & M', L, I \cdot v, \overline{(lv, v)} \\
 & & \textit{where} \\
 & & M', L, I, \overline{(lv, v)} = \texttt{match\_patt}(p, M) \\
 & & v = \texttt{foldl}(n, \lambda, L) \\
 & & \\
\text{(11) } \texttt{match\_patt}(p\ \texttt{map}(\lambda),\ M) & = & M', L, I \cdot v, \overline{(lv, v)} \\
 & & \textit{where} \\
 & & M', L, I, \overline{(lv, v)} = \texttt{match\_patt}(p, M) \\
 & & v = \texttt{map}(\lambda, L) \\
 & & \\
\text{(12) } \texttt{match\_patt}(p\ \texttt{bind}(x),\ M) & = & M', L, I \cdot (x, v), \overline{(lv, v)} \\
 & & \textit{where} \\
 & & M', L, I \cdot v, \overline{(lv, v)} = \texttt{match\_patt}(p, M) \\
 & & \\
\texttt{match\_attr}(lv \cdot \overline{at}, v \cdot \overline{v}) & = & (lv, v) \cdot \texttt{match\_attr}(\overline{at}, \overline{v}) \\
\texttt{match\_attr}(v \cdot \overline{at}, v \cdot \overline{v}) & = & \texttt{match\_attr}(\overline{at}, \overline{v}) \\
\texttt{match\_attr}(\emptyset, \emptyset) & = & \emptyset \\
 & & \\
\texttt{unify}(\emptyset, \overline{(lv, v)}) & = & \overline{(lv, v)} \\
\texttt{unify}((lv, v) \cdot \overline{(lv, v)}, \overline{(lv', v')}) & = & \texttt{unify\_single}((lv, v), \overline{(lv', v')}) \cdot \texttt{unify}(\overline{(lv, v)}, \overline{(lv', v')}) \\
 & & \\
\texttt{unify\_single}(\overline{(lv, v)}, \emptyset) & = & \overline{(lv, v)} \\
\texttt{unify\_single}((lv, v), (lv, v) \cdot \overline{(lv', v')}) & = & \emptyset \\
\texttt{unify\_single}((lv, v), (lv', v') \cdot \overline{(lv'', v'')}) & = & \texttt{unify\_single}((lv, v), \overline{(lv'', v'')}) \\
 & & \textit{if} \\
 & & lv \neq lv' \\
 & & \\
\texttt{remove}(r_n,\ p_n,\ (p_n, r_n) \cdot \overline{(p_n, r_n)}) & = & \overline{(p_n, r_n)} \\
\texttt{remove}(r_n,\ p_n,\ (p'_n, r'_n) \cdot \overline{(p_n, r_n)}) & = & (p'_n, r'_n) \cdot \texttt{remove}(r_n, p_n, \overline{(p_n, r_n)}) \\
 & & \\
\texttt{remove\_reactions}(p_n,\ (p_n, r_n) \cdot \overline{(p_n, r_n)}) & = & \texttt{remove\_reactions}(p_n, \overline{(p_n, r_n)}) \\
\texttt{remove\_reactions}(p_n,\ (p'_n, r'_n) \cdot \overline{(p_n, r_n)}) & = & (p'_n, r'_n) \cdot \texttt{remove\_reactions}(p_n, \overline{(p_n, r_n)}) \\
\texttt{remove\_reactions}(p_n, \emptyset) & = & \emptyset
\end{array}
$$

Figure 5.8 Auxiliary functions used in the reduction rules (Cont.).

us to debug and find bugs during the development of our mechanized formalization. We used and extended Redex's randomized testing suite to check if our implementation produces the result predicted by the model described in section 5.1. In the rest of this chapter, we describe this mechanized formalization and the challenges faced during its development.

A

$$
\begin{array}{rcl}
P_e \subseteq \textbf{Pattern} & ::= & \texttt{pattern}\ p_n\ \texttt{as}\ p \\
R_e \subseteq \textbf{Reaction} & ::= & \texttt{reaction}\ r_n\ \texttt{do}\ e \\
A_e \subseteq \textbf{Actor} & ::= & \texttt{actor}\ a_n\ \{\overline{\texttt{react\_to}\ p_n\ \texttt{with:}\ r_n}\} \\
e \in E \subseteq \textbf{Expr} & ::= & nil \mid x \mid l \mid i \mid \lambda\overline{x}.e \mid \{m_{id}, \overline{e}\} \mid \texttt{let}\ x = e\ \text{in}\ e \\
 & & \mid \texttt{spawn}\ a_n \mid \texttt{send}\ e, e \mid \texttt{react\_to}\ p_n,\ \texttt{with:}\ r_n \\
 & & \mid \texttt{remove}\ r_n,\ \texttt{from:}\ p_n \\
 & & \mid \texttt{remove\_reactions}\ p_n
\end{array}
$$

B

```
1 (define-language NEST
2   (pe ::= (pattern pn p))
3   (re ::= (reaction rn e))
4   (ae ::= (actor an (react-to pn rn) ...))
5   (e ::= nil
6        x l i
7        (λ [e ...] e)
8        (let (x e) in e)
9        (spawn an)
10       (send e e)
11       (react-to pn rn)
12       (remove rn pn)
13       (remove-reactions pn)
14  (x l i pn rn an ::= variable-not-otherwise-mentioned))
```

Figure 5.9 Translation of the NEST grammar to Redex

The source code of this implementation is available online[3]. A short intro to Redex and a full overview of our implementation can be found in appendices A.7 and A.8 respectively.

5.2.1 A Mechanized NEST Model

We develop the mechanized implementation of NEST in four phases.

- First, we translate the grammar of our paper-and-pencil model into Redex using the primitive `define-language` (see figure 5.9).

[3]NEST Implementation - `https://github.com/rhumbertgz/nest-plt-redex`

Listing 5.1 Examples of NEST grammar tests in Redex

```
1  (module+ test
2    (define valid-pe-exp? (redex-match? NEST pe))
3
4    (define p1 (term (pattern pn_1 (:indoor_humidity a))))
5    (define p2 (term (pattern pn_2 ((:motion a) (count 4)))))
6    (define p3 (term (pattern pn_3 ((and (:indoor_humidity a)
7                                         (:outdoor_humidity b))
8                                         (when (> a b))))))
9
10   (test-equal (valid-pe-exp? p1) #true)
11   (test-equal (valid-pe-exp? p2) #true)
12   (test-equal (valid-pe-exp? p3) #true)
13
14   (test-results)
15  )
;; All 3 tests passed.
```

- Second, we test each expression of the Redex model using handwritten (see listing 5.1) and randomized tests (see listing 5.2).
- Third, we translate the reduction rules of our paper-and-pencil model into Redex using the primitive `reduction-relation` (see listing 5.4).
- Fourth, we test each reduction rule of the Redex model using handwritten (see listing 5.1) and randomized tests (see section 5.2.2).

Figure 5.9 shows the grammar described in section 5.1.1 (A) and its corresponding definition in Redex (B). Both definitions looks almost the same, except that in line 7 we uses three dots (...) to express one or more expressions e, and then in line 14 we use the built-in production `variable-not-otherwise-mentioned` to match any of variable used in the productions of the above non-terminals. Figure 5.9.B excludes other non-terminals required to support the different definitions of p (see figure 5.2). However, the full definition of the language can be found in appendix A.8.

Redex provides language designers with a test suite to test their models. Listing 5.1 shows three hand-written tests for pattern expressions `pe`. This example defines a module `test` with four main blocks of code. Line 2 shows the definition of a helper function that uses the primitive `redex-match?` to

Listing 5.2 Examples of NEST grammar randomized tests in Redex

```
1  (module+ test
2    (define valid-pe-exp? (redex-match? NEST pe))
3
4    (redex-check
5      NEST
6      pe
7      (valid-pe-exp? (term pe))
8      #:attempts 10000)
9   )
  ;; redex-check: ../nest-plt-redex/random-tests1.rkt:18
  ;; no counterexamples in 10000 attempts
```

determine if a particular term matches the non-terminal `pe` of the NEST language. Lines 4-8 define three pattern expressions where the `pn_1` will match any message of type `:indoor_humidity`, `pn_2` will match upon the reception of four messages of type `:motion`, and `pn_3` will match upon the reception of a message of type `:indoor_humidity` and another of type `:outdoor_humidity` if the value of the indoor humidity (`a`) is greater than the value of the outdoor humidity (`b`). Lines 10-12 check if the above patterns match a valid production of `pe`. Finally, line 14 prints the results of the tests (see the last line) by invoking the built-in function `test-results`.

Hand-written tests were helpful for initial checks of NEST's syntax. However, they require extra work, and they can not exhaustively test whether NEST is a *solid* formal model that has no bugs. To overcome this limitation, we also defined random tests to check syntactic properties of the NEST model using Redex's *randomized* test suite. For example, listing 5.2 shows an example of such random checker for the non-terminal `pe`. The first two lines of this example are identical to the ones of listing 5.1. Nevertheless, lines 4-8 use Redex's primitive `redex-check` try to find a randomly generated term that does not satisfy the condition on line 7 (a.k.a. *counterexample*). The first two arguments of `redex-check` represent the language (line 5) and the type of expression that it should generate (line 6), respectively. The last argument (line 8) sets the number of attempts (by default 1000). The last two lines of this listing show an example of the output printed after the execution of `redex-check`.

Listing 5.3 Add evaluation contexts to NEST

```
1 (define-extended-language NEST-R NEST
2  (E ::=
3     hole
4     (v ... E e ...)
5     (let (x E) in e)
6     (send E e)
7     (send v E)))
```

The third phase of the NEST mechanized implementation requires translating the reduction rules defined in section 5.1.3 into Redex. In order to support the test of such reduction rules, we have to add support for evaluation contexts to the Redex model of NEST defined in figure 5.9. Listing 5.3 exemplifies the extension the initial NEST model to add a four of evaluation contexts by means of the of new non-terminal E. An evaluation context is a special term that contains a `hole`. Reduction relations can match a term against an evaluation context to find a sub-term that matches the hole.

Listing 5.4 Definition of the `react-to` reduction rule in Redex

```
;;=============================================
;; Extracted from the function NEST-Reductions
;; Source file: nest-reductions.rkt
;;=============================================

134 [ -->
135  (in-hole K
136          (pl
137           rl
138           (actor id q pr (in-hole E (react-to pn_a rn_b)))))
139  ,(term-let
140    ([p (term pn_a)] [r (term rn_b)])
141      (term
142       (in-hole K
143               (pl
144                rl
145                (actor id q (ADD-REACTION pr p r) (nil))))))
146  "react-to"]
```

Listing 5.4 exemplifies the translation of the reduction rule `react-to` defined in section 5.1.3 on page 71. In Redex, reduction rules are defined by the operator `-->` (line 134). The first argument (lines 135-138) of this operator, searches a term that contains the subterm `(react-to pn_a rn_b)`. The second argument (lines 139-145) reduces that subterm and puts a new value back into the hole in the evaluation context. The auxiliary function `ADD-REACTION` (line 145) updates the list of pattern-reaction bindings list. The third argument specifies the rule's name, `react-to`.

Similarly, to regular syntax tests, Redex provides language designers with the operator `test-->>` to test reduction of rules. Listing 5.5 shows a unit test that checks the correct reduction of the `react-to` rule. The operator `test-->>` checks if the first term (lines 127-129) reduces to the second one (lines 130-132). In this case, the expression on line 129, and particularly the term `(react-to p1 r1)` must be reduced. Line 32 shows how the actor with id `a_40` ends in a new state where it has a new pattern-reaction binding pair, and the `react-to` expression was reduced to `nil`. This example also showcases both global pattern and reaction maps (lines 127-128, 130-131).

Listing 5.5 Test example for the reduction rule `react-to`

```
;;=======================================
;; Extracted from function test-reactions
;; Source file: nest-reductions.rkt
;;=======================================

126 (test-->> NEST-Reductions
127    (term ((((p1 . (((:humidity x_in x_out) nil) ()))) 
128            ((r1 . ("Firing reaction 1")))
129            (actor a_40 () () (react-to p1 r1)))))
130    (term ((((p1 . (((:humidity x_in x_out) nil) ()))) 
131            ((r1 . ("Firing reaction 1")))
132            (actor a_40 () ((p1 . (r1))) (nil))))))
133
```

Redex's test suite also supports the visualization of the reduction graph of a particular term using the function `traces`. For example, listing 5.6 traces a term where a binding between the pattern `p1` and the reaction `r1`

Listing 5.6 Trace the reduction process of a term

```
1 (traces NEST-Reductions
2   (term ((() () (actor root_7 () ()
3             (new-actor (~
4                 (pattern p1 (:humidity x_in x_out))
5                 (reaction r1 "Firing reaction 1")
6                 (react-to p1 r1)))))))))
```

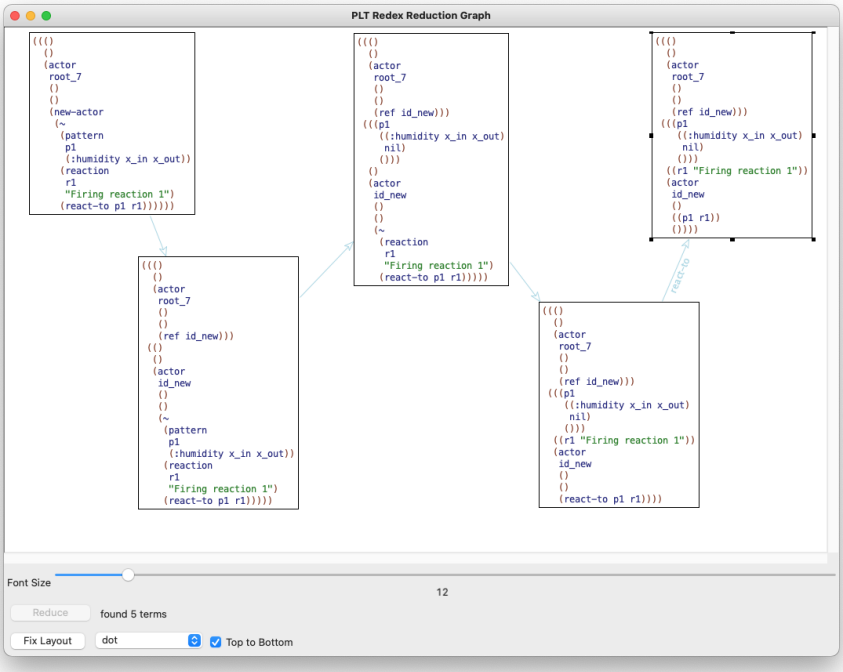


Figure 5.10 Screenshot of a term's reduction graph using the traces primitive

is added to a new actor. Figure 5.10 shows the resulting reduction graph of listing 5.6.

5.2.2 Randomized Tests of NEST's Patterns

NEST's patterns introduce new challenges to validate the correctness of the mechanized model of NEST. Patterns only match when a message or a list of messages satisfy predefined conditions set by their guard expressions and operators (e.g., `count`, `windows`, `sequencing`). However, such constraints cannot be satisfied by Redex's randomized tests. These tests only generate random instances of a non-terminal in an attempt to falsify it. Additionally, a pattern may have multiple reaction's bindings. The process of executing the reactions of a pattern after a complete match is called *pattern activation*. To evaluate the correctness of a pattern, we need to check that it had the right number of activations for a particular list of messages. For instance, if a list of 20 random *valid* messages is generated for an elementary pattern with a single reaction, 20 activations should be expected. In contrast, if a list of 20 random *invalid* messages is generated for the same elementary pattern, zero activations should be expected instead. Therefore we state the correctness of a NEST's pattern test if:

- The total number of activations of the pattern equals the total number of valid message sequences generated by the test.
- The total number of activations of the pattern is zero after invalid message sequences were generated by the test.
- The inbox of an actor is empty by the end of a test no matter if valid or invalid message sequences were generated.

To address the limitations of Redex's test suite for NEST's patterns, in this section, we introduce a proposal for randomized tests of NEST's patterns. Briefly, we extended Redex's test suite with two functions: `pattern-test` and `pattern-traces`. The former aims to prove the correctness of NEST's patterns using a randomized testing approach particularly designed for them. The latter targets to ease debugging sessions by visually representing the reduction graph of a NEST program.

5.2.2.1 pattern-test

Listing 5.7 shows an example of use of the new `pattern-test` function. This function takes as arguments a reduction relation (line 2), a pattern term (line 3-4), and three optional arguments: the number of iterations

Listing 5.7 Example of a NEST pattern test

```
1  (pattern-test
2     NEST-T-Reductions
3     #:pattern (term (pattern p1 ((:humidity x_in x_out)
4                                  (when (< x_in x_out)))))
5     #:iterations 4000
6     #:log-output 'basic
7     #:polluted-msgs #false)
8
9  (pattern-test-results)
   ;;============================================================
   ;; pattern-test: (pattern p1 ((:humidity x_in x_out)
   ;; (when (< x_in x_out))))
   ;; - random messages: 4000
   ;; - matched messages: 4000
   ;; - fired reactions: 4000
   ;;============================================================
   ;; All pattern tests passed.
```

(line 5), the log level (line 6), and a boolean to specify if the messages should be polluted or not (line 7). The number of iterations represents the total of message sequences that must be generated. At the end of its execution, this function will check the correctness of the pattern based on the conditions defined at the beginning of section 5.2.2. For example, if the `#:polluted-msgs` argument is not specified the number of activations (i.e., the total of fired reactions) must be equal to the number of iterations passed to the function. Internally, this function creates an actor, and it binds the pattern received as an argument to an auto-generated reaction. Conversely, if the `#:polluted-msgs` argument is set to `#true`, the pattern should **not have** any activation. This function also checks that the actor's inbox must be empty at the end of each iteration. If any above conditions are not met, the test fails.

The optional arguments of the function `pattern-test` have the following default values: `#:iterations = 1000`, `#:log-output = 'none`, and `#:polluted-msgs = #false`. The `#:log-output` argument accepts three values `'none`, `'basic` and `'advanced`. In its basic log setting the `pattern-test` function shows only general statistics (see listing 5.7). If the advanced log

Listing 5.8 Trace the reduction process of a pattern

```
1 (pattern-traces
2     NEST-T-Reductions
3     #:pattern (term (pattern p1 ((:humidity x_in x_out)
4                                  (count 2 )
5                                  (when (> x_in x_out))))))
```

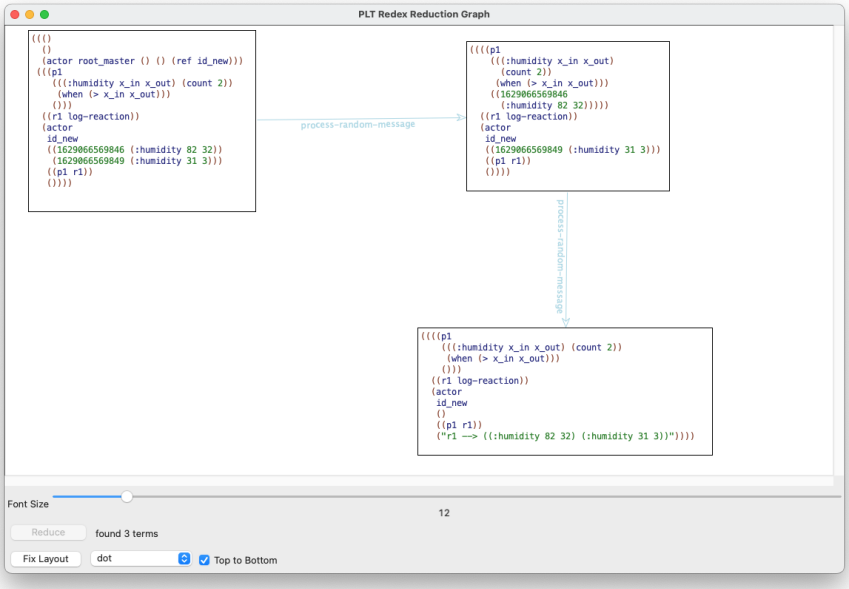


Figure 5.11 Example of a reduction graph using the pattern-traces function

setting is enabled, the list of generated, matched and consumed messages by each activation will be printed.

The call to the function `pattern-test-results` (line 9) will print out on the terminal how many tests passed and failed. Furthermore, it resets the counters so that the next time this function is called, it prints the results for the next round of tests.

Listing 5.9 A simple NEST pattern test

```
1 (pattern-test
2     NEST-T-Reductions
3     #:pattern (term (pattern p1 ((:humidity x_in x_out)
4                                  (when (< x_in x_out)))))
5     #:iterations 4)
```

5.2.2.2 pattern-traces

The function `pattern-traces` is a simplified version of `pattern-test` for visualization and debugging purposes. It only generates one round of messages and does not show any logs, nor validates any of the conditions checked by `pattern-test` (e.g., number of activations). However, it can visualize reduction graphs for patterns with polluted and non-polluted messages (see listing 5.8). Figure 5.11 shows the reduction graph for the traced pattern defined in listing 5.8. In this figure can be observed how the pattern's reaction consumes the list of generated list messages.

5.2.2.3 Randomized Pattern Test Generation Algorithm

This section describes the methodology used *to generate random messages* used by the `pattern-test` and `pattern-traces` functions. Redex's randomized terms are limited to a particular language expression However, NEST abstractions require a more refined random strategy taking into account the structure and constraints of a pattern.

Listing 5.10 exemplifies NEST randomized pattern test methodology applied to the implementation of the `pattern-test` function. In order to facilitate the explanation of this algorithm we will use the example shown in listing 5.9. The function `pattern-test` (see listing 5.10) executes the following operations:

- Define an expression `actor_def` that uses the pattern `p` as the only *message pattern* of the interface of a new actor (line 7). Additionally this expression includes the binding of a reaction `r1` to the pattern `p` (`p1` in our example of listing 5.9). The body of the reaction `r1` only prints the list of matched messages. For example, the pattern defined in listing 5.9 will be transformed into the following term:

```
(new-actor (~ (pattern p1 ((:humidity x_in x_out)
                           (when (< x_in x_out))))
              (reaction r1 log-reaction)
              (react-to p1 r1)))
```

- Reduce the expression `actor_def` until all its subterms cannot be reduced further (line 8). In this case, a new actor will be created with a binding between its only pattern and its only reaction. The resulting `actor` term is listed below.

  ```
  (actor id_new () ((p1 r1))
  ```

  This term represents a new actor with identifier `id_new`, an empty inbox `()`, and its pattern-reaction bindings `((p1 r1))`. Remember that patterns and reactions are stored in global scope (see section 5.1), for this reason we omitted their representation.

- Decompose the main components of the pattern `p` (see section 4.3) into a map (line 9). For example, the pattern `p1` shown in listing 5.9 will be decomposed into a map `p_parts`, where the `selector` key's value is `(:humidity x_in x_out)` and the `guard` key's value is `(< x_in x_out)`.

- Generate valid or non-valid messages sequences for pattern `p` (line 10). For example, the test defined in listing 5.9 requires *four* messages sequences of length 1, since `p1`'s selector matches *single* messages. The number of iterations `n` passed to the function `pattern-test` determines the total of messages sequences to generate. In case of a composite (e.g., `(pattern pn_1 (and (:msg1 a b) (:msg2 c d))))`) or an accumulation pattern (e.g., `(pattern pn_2 ((:msg2 4) (count 4)))`), the generated messages sequences will contains the required messages by the pattern. If the value of `pollute` argument is `#true` none of the messages sequences should match pattern `p`. Internally the `build-message-sequences` function executes the following actions:

  1. Extract all existing logic variables in the selectors of the pattern `p` and store them in a list. For example, the value of this list for the pattern `p1` defined in listing 5.9 will be `(x_in, x_out)`.

2. Collect the constraints of all logic variables and store them in a map. For example, the pattern *p1* has a guard expression that constrains the value of *in*. In this case, the value of `x_in` must be less than the value of `x_out`.
3. Sort the list logic variables based on their guard constraints. Logic variables without constraints will be moved to the end of the list. In contrast, logic variables with at least a comparison against a particular value (e.g., `(> x 5)`) will be moved to the beginning of the list. At the same time, logic variables' constraints are also sorted. Constrains which compare a variable against a particular value will be moved to the beginning of the list of constraints. Just like Sparrow and Elixir, NEST only allows a limited set of guard expressions, and custom user functions are not allowed. The current implementation of NEST only supports numeric comparisons in guard expressions. However, NEST pattern-matching capabilities allow a pattern's selector to match any value it supports. For example, the following selector `(:humidity v 'bedroom)` will match messages of type `:humidity` that its second attribute (`location`) is equals to `'bedroom`.
4. Generate a map of valid or non-valid random values for each logic variable. If a logic variable is involved in a guard, its value will always be numeric, and it must satisfy its guard constraints. Otherwise, the value of the logic variable can be a string, boolean, atom, or a number. During the generation of messages, this map is used to guarantee that shared logic variables between the selectors of a pattern get assigned the same value.

- Process each of the messages sequences generated. This task is executed in three steps:

1. Inject the messages `ml` into the inbox of the `actor` term for further reduction of this term (line 14).
2. Reduce the `new_term` that contains the injected messages (see line 15).
3. Check if the injected messages matched the pattern (line 18). If the actor's inbox is empty, the next messages sequence will

be processed. Otherwise, an error will be raised, and the test will fail (line 19).

Listing 5.10 Implementation of the `pattern-test` abstraction

```
1  (define
2    (pattern-test r #:pattern p #:iterations [n 1000]
3                  #:polluted-msgs [pollute #false]
4                  #:log-output [level 'none])
5
6      (let*
7        ([actor_def (build-base-term p)]
8         [actor (reduce-term r actor_exp)]
9         [p_parts (decompose-pattern p)]
10        [msgs (build-message-sequences p_parts n pollute)])
11
12          (for-each
13              (lambda (ml)
14                (let* ([new_term (inject-messages actor ml)]
15                       [new_actor (reduce-term r new_term)]
16                       [inbox (get-actor-inbox new_actor)])
17                      (cond
18                        [(empty? inbox) 'continue ]
19                        [else raise "Invalid reduction."])))
20               msgs)
21         (cond
22           [(check-activations n pollute)
23            (print-logs p msgs level)]
24           [else raise "Invalid reduction."])))
```

- Compare the number of pattern activations against the number of iterations (line 22). Remember that the generated actor has a single pattern and a single reaction. The total of activations should be equal to the number of iterations if the generated messages were valid. Contrary, the number of activations should be zero if the messages were polluted. The test succeeds if the previous condition is met. Then any required output log will be printed (line 23). If the test fails, an error will be raised (line 24).

As mentioned in section 5.2.2.2, `pattern-traces` is a simplified version of `pattern-test` designed only for visualization and debugging support.

Listing 5.11 shows the implementation details of this function. The implementation of `pattern-traces` differs from the one of `pattern-test` in three main aspects. First, it has a reduced number of arguments (lines 2-3).

Listing 5.11 Implementation of the `pattern-traces` abstraction

```
1  (define
2    (pattern-traces r #:pattern p
3                    #:polluted-msgs [pollute #false])
4      (let*
5         ([actor_def (build-base-term p)]
6          [actor (reduce-term r actor_exp)]
7          [p_parts (decompose-pattern p)]
8          [msgs (build-messages-sequence p_parts pollute)]
9          [new_term (inject-messages actor msgs)])
10
11        (traces r new_term)))
```

Second, it requests the generation of a single messages sequence (line 8). Third, it does not check for an empty inbox nor the number of activations. After it injects the generated messages into the actor's inbox (line 9), it passes the resulting term to the Redex's primitive `traces` to visualize its reduction graph (line 11).

5.2.3 NEST compared to Sparrow

In this section, we list the differences between NEST's formal semantics and the actual implementation of Sparrow as explained in chapter 4. Our goal with NEST was never to encompass a fully-fledged implementation of Sparrow. Instead, we only formalised the core parts of Sparrow. These parts cover the semantics of Sparrow programs in the interest of their formal reasoning. For this reason, a few Sparrow abstractions were purposefully omitted from NEST. In a future avenue of our research NEST may support such Sparrow abstractions. Here we list the differences between NEST and Sparrow.

- NEST is built conceived atop a base language, which is not included in our formalization. Sparrow is built on top of Elixir as a domain-specific language. The base language of NEST is limited to a subset of Elixir: only primitive values and comparison/logic operators are used in our formalization.

- In NEST the `debounce` and `interval` operator implemented in Sparrow are not formalized. Only the `window` operator can be used to declare *unquantified accumulation patterns*.
- NEST patterns cannot combine quantified (e.g., `count`) and unquantified (e.g., `window`) operator at the same time. However, Sparrows supports the particular combination of `count` and `window` operators in its patterns.
- NEST does not formalize negated patterns.
- NEST formalizes two of the three filter mechanism for messages implemented in Sparrow: *pattern-matching* and *regular guards*. In contrast, Sparrows supports *inline guards* which are a syntactic sugar to write more compact filter expressions.
- NEST does not formalize advanced attribute operators (e.g., *alias*, *may be equal*, and *must be equal*) that are used for the definition of collection patterns.

Although NEST does not implement all Sparrow's abstractions, its current implementation is fully functional and serves its experimental purpose.

5.3 Conclusion

The NEST formalism plays a vital role in the precise description of Sparrow's coordination abstractions. Our mechanized implementation of NEST in Redex allowed us as language designers to validate its grammar and reduction rules. We defined an extensive set of handwritten and randomized tests based on Redex's test suite to accomplish such validation. However, we could not use the standard randomized test suite provided by Redex to exhaustively test NEST *patterns*. To satisfy the requirements imposed by such patterns, we implemented a randomized message generator that can provide a list of valid or invalid messages for a particular pattern. We used this generator as the backbone of two randomized testing abstractions for NEST *patterns*. These abstractions allowed us to validate the correct results of the evaluation of NEST reduction relations with randomized messages. Future work may entail the implementation of Sparrow' abstractions missing in our mechanized model.

CHAPTER 6

Sparrow: An Elixir DSL Implementation

Up to now, we have described the formal aspects of Sparrow. In this chapter, we describe the details of its implementation as an Elixir domain-specific language based on macros. Section 6.1 introduces the strengths and weaknesses of Elixir's macro system. Section 6.2 presents a detailed description of low-level design decisions that we made for both syntax and internal implementation of Sparrow's actors. Section 6.3 describes the design and implementation of a message pattern engine. Finally, section 6.4 describes the implementation of two language tooling support for Sparrow-based applications.

6.1 DSLs in Elixir

Elixir allows developers to tailor their applications to a particular domain through domain-specific languages (DSLs). An Elixir's DSL can be built using a combination of data structures, functions, and macros. In this section, we will focus on the latter approach since it was the main mechanism used in the implementation of Sparrow's language constructs. To facilitate its extensibility, Elixir provides two main metaprogramming tools.

- First, it exposes the abstract syntax tree (AST) of its programs using its own data structures. This help developers to seamlessly interact

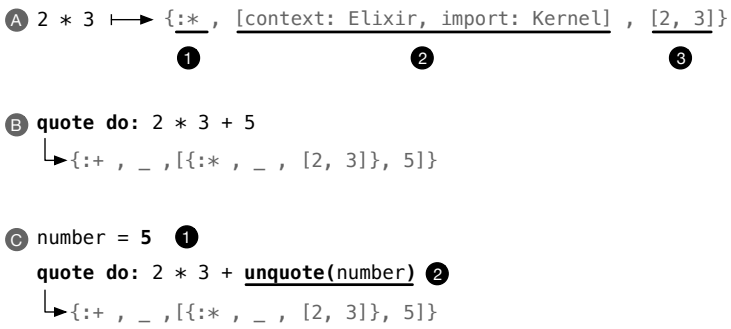


Figure 6.1 Metaprogramming tools in Elixir

with the syntax of their programs. Every expression breaks down into a three-element tuple in the AST. For example, figure 6.1A showcases the above homoiconic behaviour. The AST of the expression (`2 * 3`) on the left is represented as the three-element tuple on the right. In this example, A.1 is an atom[1] denoting a function call (`*` operator), but it can be another tuple representing a nested node in the AST. A.2 represents metadata about the expression. In this case, the operator `*` is just syntactic sugar for the `Kernel.*` function. Finally, A.3 holds the list of arguments for the function call.

- Second, Elixir provides a clear macro language providing syntax shortcuts for `quote` and `unquote` expressions. These expressions use a high-level syntax to receive ASTs as arguments and provide ASTs as return values. At the same time, they give control to developers to extend the language. Elixir itself extensively uses them. For example, the constructs `defmodule`, `def`, and `receive` used in listing 6.1 are all macros. In a similar way, other Elixir's top-level constructs like `if`, `case`, and `cond` are implemented as macros. Figure 6.1B illustrates the simplicity of transforming a high-level source to its low-level AST using the `quote` macro. For reasons of brevity, we have omitted the *metadata expression* of the resulting AST, and instead used the _ character. In contrast, figure 6.1C demonstrates the injection of an outside bound variable (C.1) into an AST using the `unquote` macro (C.2).

[1] Atoms are constants whose values are their own name.

Listing 6.1 A counter actor in Elixir

```
1  defmodule Counter do
2
3    def start() do
4      spawn fn -> listen(0) end
5    end
6
7    def listen(count) do
8      receive do
9        :inc ->
10         listen(count + 1)
11       {:val, sender} ->
12         send sender, count
13         listen(count)
14     end
15   end
16
17 end
```

In summary, Elixir allows DSL developers to manipulate and inspect the AST of their programs in a way typically reserved only for compilers and language designers. Besides the above-mentioned features, Elixir's macros can also limit the expressiveness of its DSLs. The next sections will summarize both strengths and weakness of Elixir's macros.

6.1.1 Macros: the good

In this section, we describe four main properties of Elixir macros.

- First, macros are *hygienic* by default. Variables defined inside of a macro would not conflict with variables defined in the context where that macro is expanded (*user's code*). In the same way, alias definitions and function calls available in the macro context are not going to leak into the user context. However, under specific situations, hygiene can be bypassed[2].

[2] `https://hexdocs.pm/elixir/Kernel.html#var!/2` [Accessed: 04-11-2020]

- Second, macros are *lexical scoped*. A macro cannot inject code or other macros globally. The developer has to require or import the module that defines the macro explicitly.

- Third, macros are *explicit*. It is impossible to run a macro without an explicit invocation. In other words, macros must be explicitly called in the caller context at compilation time.

- Fourth, macros can be *public* or *private*. Private macros are only available at compilation time inside the module that defines them. In both cases, developers must define macros before their usage; otherwise, their invocation will raise an error at runtime.

Sparrow's macros are public and they are explicitly called in the context of an instance of its `Actor` module.

6.1.2 Macros: limitations

Although Elixir provides developers with a powerful macro language, its macros have certain limitations to create *new operators*. This section identifies such limitations and describes some alternatives used in this dissertation to circumvent them.

In Elixir, unlike Haskell, macros cannot define new operators. This limitation forces the shape of Sparrow's syntax (see figure 6.2). For example, figure 6.3B shows a composite pattern using an ideal `andThen` operator to detect a sequence of messages. However, due to the above limitation our real implementation (see figure 6.3C) had to use the standard `and` operator together with a map to set the desired sequencing behaviour (see the highlighted code in gray color). Similarly, other syntaxes such as the ones to specify timing constraints (see section 4.3.2.4), message selection strategy (see section 4.3.2.5), and message accumulation (see section 4.3.3) were affected. Figure 6.3E shows another example of the above syntax limitations. In summary, the limitation to define new operators forced us as language designers to add an extra layer of square brackets and commas to the syntax of Sparrow's patterns.

To overcome the above-mentioned limitations, Sparrow rewrites the AST to change the meaning of valid Elixir expressions (e.g., `as`, `when`) for its patterns definitions. Furthermore, it overrides operators (e.g., ~>)

```
<p-definition>      := pattern <identifier> as <pattern>
<r-definition>      := reaction <identifier> (<arg>, <arg>, <arg>) do <expression> end
<react-to>          := react_to <identifier>, with: <identifier>
<remove-from>       := remove <identifier>, from: <identifier>
<remove-reactions>  := remove_reactions <identifier>
<pattern>           := <comp-pattern> [<guard>] [, [options: <option>+]]
<comp-pattern>      := <elem-pattern> [(and | or ) <elem-pattern>]*
<elem-pattern>      := [not] <selector> [[<operator>+]] {|> <transformer>}*
<selector>          := {<symbol>, <attribute>*}
                     | <identifier>[{{<inline-guard>
                     | <alias-op>}+}]
<attribute>         := <value>
                     | <symbol>
                     | <logic-var>
<guard>             := when <expression>
<inline-guard>      := <identifier> = <expression>
<alias-op>          := <identifier> ~> <identifier>
<symbol>            := :<identifier>
<logic-var>         := [(@ | ! )]<identifier>
<operator>          := window: <time>
                     | debounce: <time>
                     | every: <number>
                     | count: <number>
<transformer>       := fold(<expression>, <expression>)
                     | bind(<identifier>)
<option>            := seq: <boolean>
                     | interval: <time>
                     | last: <boolean>
<time>              := {<number>, (:secs | :mins | :hours | :days | :weeks)}
<arg>               := <identifier>
```

Figure 6.2 Sparrow EBNF-styled syntax (repetition of figure 4.1)

parsed but not used by Elixir[3]. By using this approach, Sparrow patterns do not require an entirely new syntax on top of Elixir.

6.2 Sparrow Actors

In this section, we describe the implementation details behind the multiple-message match interface of Sparrow's actors. These actors differ from traditional Elixir ones in three aspects.

- First, their interface is defined by patterns that can match multiple messages, instead of individual messages (see section 4.3).

[3] `https://hexdocs.pm/elixir/operators.html` [Accessed: 04-11-2020]

```
A pattern motion_sensor as {:motion, id, :on, location}

B pattern occupied_home as motion_sensor{location= :front_door}
                  andThen {:contact, id, :open, :front_door}
                  andThen motion_sensor{location= :entrance_hall}

C pattern occupied_home as motion_sensor{location= :front_door}
                     and {:contact, id, :open, :front_door}
                     and motion_sensor{location= :entrance_hall},
                     options: [ seq: true ]

D pattern heating_failure as {:failure, id, @code} count 3

E pattern heating_failure as {:failure, id, @code}[count: 3]
```

Figure 6.3 Limitations to define new operators: (A) Define an elementary pattern; (B) Define a composite pattern using the fictional `andThen` sequencing operator; (C) Real implementation of the sequencing operator in Sparrow; (D) Definition of a quantified accumulation pattern using a fictional syntax of a `count` operator; (E) Real implementation of a quantified accumulation pattern in Sparrow

- Second, they have a *virtual inbox* that it is used by an embedded *message pattern engine* (see section 6.3). After an actor receives a message, it is forwarded to its virtual inbox and the matching process starts.

- Third, messages in the virtual inbox have a *finite lifetime* established by its actor. Subsequent a message expires, it is automatically garbage collected by the actor's message pattern engine.

Listing 6.2 shows the essential parts of the `Sparrow.Actor` module implementation. We use three dots (...) and comments (#) to denote code excluded from the listing. The full source code of this module can be found in the appendix A.2. Sparrow's actors are implemented as an extension of the `GenServer`[4] behaviour (line 7). A behaviour provides a way to define a set of functions that have to be implemented by a module. Furthermore, it ensures that a module implements all the functions in that set.

[4] `https://hexdocs.pm/elixir/GenServer.html` [Accessed: 06-11-2020]

Listing 6.2 Implementation of Sparrow's actor module

```
defmodule Sparrow.Actor do
  ...
  defmacro ___using___(_) do
    ...
    quote do
      import Sparrow.Actor
      use GenServer

      ## Actor API
      def start(options \\ [], linked \\ true), do: # code
      def stop(pid), do: # code
      def send(pid, message), do: # code
      ...
      ## Actor Callbacks
      @impl true
      def handle_continue(:init, args) do
        state = Sparrow.Actor.___init(___MODULE___, args)
        {:noreply, state}
      end
      @impl true
      def handle_cast({:send, msg}, {engine, _}= state) do
        Jupiter.process_message(engine, msg)
        {:noreply, state}
      end
    end
  end

  defmacro pattern({name,_, [{:as,_, [p | []]}]}) do
    ...
  end
  ...
  defmacro reaction({r_name,_, [header]}, do: body) do
    ...
  end
  ...
  defmacro react_to({p_name,_,_}, with: {r_name,_,_}) do
    ...
  end
  ...
  defmacro remove({r_name,_,_}, from: {p_name,_,_}) do
    ...
  end
  ...
  defmacro remove_reactions({p_name,_,_}),do
    ...
  end
  ...
end
```

Like any other Elixir behaviour, `Sparrow.Actor` consists of a generic part (*Actor API* lines 10-12) and a specific part (*Actor callbacks* lines 14-24). The former provides basic actions such as *start* (line 10) and *stop* (line 11) an actor, as well as *send* an asynchronous message (line 12). The latter shows the the definition of two callbacks functions `handle_continue` and `hande_cast`. The first callback is asynchronously invoked during the initialization of the actor after invoking the `start` function. This callback is responsible for initializing the actor's message pattern engine (line 17). The first one is a standard `GenServer` callback to handle asynchronous messages. In this case, the *handle_cast* function will match a message `{:send, msg}` and forward its body (*msg*) to the actor's message pattern engine for its further processing. In both cases, the actor's API and callback functions are injected to an `Sparrow.Actor` instance using the macro `___using___`. Additionally, the actor module defines a set of macros to build *patterns* and *reactions* expressions (lines 28-47) which form part of the actor behaviour API.

Listing 6.3 exemplifies the definition and instance of an actor in Sparrow. Lines 1-10 define an actor `BedRoomActor` which will turn on the bedroom's light. In line 2 we extend the `Sparrow.Actor` module through the macro `use`. This macro invokes the `Sparrow.Actor.___using___` macro to inject its body in the new module `BedRoomActor`. At the same time, it makes accessible to the `BedRoomActor` module all macros defined by `Sparrow.Actor`. In this example, the interface of this actor consist of a single *motion* pattern that only match messages of type `:motion` if its second and third attributes have values `:on` and `:bedroom` respectively (line 4). In line 6, we define a *turn_on_light* reaction which body have been omitted for brevity. Later, in line 8, we bind both pattern and reaction using the `react_to` macro. On the other hand, the module *HomeManager* (lines 12-21) initializes an instance of the `BedRoomActor` using the `start` function (line 15). Next, in lines 16-18, we send three messages to the new actor using its id reference (`pid`). By running this program on a terminal we can verify that only the first message (lines 16) was matched and consumed by the *turn_on_light* reaction. The full implementation of this example can be found in **??**.

6.2.1 Message Patterns

As we mentioned in section 6.1, Sparrow's patterns are implemented as macros. Since macros receive the AST representation of their arguments,

Listing 6.3 Definition and instance of a Sparrow's actor

```
1  defmodule BedRoomActor do
2    use Sparrow.Actor
3
4    pattern motion as {:motion, id, :on, :bedroom}
5
6    reaction turn_on_light(l, i, t), do: # send on command
7
8    react_to on_motion, with: turn_on_light
9
10 end
11
12 defmodule HomeManager do
13
14   def run do
15     {:ok, pid} = BedroomActor.start
16     BedroomActor.send pid, {:motion, 301, :on, :bedroom}
17     BedroomActor.send pid, {:motion, 301, :off, :bedroom}
18     BedroomActor.send pid, {:motion, 303, :on, :kitchen}
19   end
20
21 end

   # Run the program
   iex> HomeManager.run
```

we exploit Elixir's pattern-matching features to determine which pattern definition was invoked. For example, in listing 6.4, we define three `pattern` macros that match different ASTs (lines 4-12). The first one (line 4), matches patterns without options (e.g., *sequencing*, *interval*). The second one (line 8), will match only patterns with options. The last one (line 12), will match any pattern definition that did not match by its predecessor and will raise an exception at compile time. In this case, the symbol_ is used to ignore a given value of the AST. All these macros, except the last one, will inject a new function to the module instantiating the `Sparrow.Actor` module. Such a function will return a struct value of type `Sparrow.Core.Pattern`. This struct can be seen as an object of traditional object-oriented languages (e.g., Java) where its properties (fields) store all constraints and operators of the defined pattern. During the actor

Listing 6.4 Pattern macros

```
1  defmodule Sparrow.Actor do
2    ...
3
4    defmacro pattern({name,_, [{:as,_,[p | []]}]}) do
5      Builder.build_pattern(name, p)
6      |> Macro.expand(___CALLER___)
7    end
8    defmacro pattern({name,_, [{:as,_,[_,_ | []]=p}]}) do
9      Builder.build_pattern(name, p)
10     |> Macro.expand(___CALLER___)
11   end
12   defmacro pattern(_), do: # raise an error
13
14 end
```

initialization, the pattern definitions are retrieved to build the actor's message pattern engine.

To facilitate the maintenance, debugging, and testing of Sparrow's macros, we avoided injecting a large amount of code into them. We split their definition and the transformation process, by using regular functions to do the transformation work. For instance, the macro definition in lines 4-7 only does two actions: matches a particular pattern AST (line 4) and expands its output AST in the caller's context (line 6). The auxiliary Builder module (line 5) generates all the code related to the pattern declared by the developer. All macros introduced in this chapter follow the above two-step procedure. The last instance of each overloaded pattern macro (line 12) will match any other AST not matched by its predecessors. In this case, an error will be raised and the compilation process will fail.

6.2.2 Pattern Reactions

Reactions resemble an Elixir function which always receives three arguments. However, we opted for a macro implementation instead of a plain Elixir function to enforce this constraint at compile time. Lines 3-7 show the definition of the *reaction* macros. The first macro pattern matches the AST of a reaction expression (line 4). Once again, the `Builder` helper module is in charge of the required transformations. The function

Listing 6.5 Reaction macros

```
1  defmodule Sparrow.Actor do
2    ...
3    defmacro reaction({name, _, [header]}, do: body) do
4     Builder.build_reaction(name, header, body)
5     |> Macro.expand(___CALLER___)
6    end
7    defmacro reaction(_, _), do: # raise an error
8
9    defmacro react_to({p_name, _, _},
10                      with: {r_name, _, _})
11                      when is_atom(p_name)
12                       and is_atom(r_name) do
13     Builder.register_react_to(p_name, r_name)
14     |> Macro.expand(___CALLER___)
15   end
16   defmacro react_to(_, _), do: # raise an error
17
18   defmacro remove({r_name, _, _},
19                   from: {p_name, _, _}) do
20     Builder.remove_reaction(r_name, p_name)
21     |> Macro.expand(___CALLER___)
22   end
23   defmacro remove(_, _), do: # raise an error
24
25   defmacro remove_reactions({p_name, _, _}) do
26     Builder.remove_reactions(p_name)
27     |> Macro.expand(___CALLER___)
28   end
29   defmacro remove_reactions(_), do: # raise an error
30
31 end
```

`build_reaction` returns the AST of a new function with the same name as the reaction containing the reaction's body. Finally, the resulting AST is injected in the caller's context by calling the `Macro.expand` function (line 5). The implementation of the macros to bind and unbind reactions to patterns was done similarly (lines 9-29).

6.3 JuPITer: A Pattern Detection Engine for Sparrow

Patterns and reactions patterns cover the syntax to define the interface of Sparrow's actors. The computational logic to match the messages required by an actor's patterns is implemented by an embeded*pattern detection engine*, called JuPITer (Join PaTterns Engine).

JuPITer builds a directed graph (also known as *discrimination network*) representing all patterns defined in an actor. For example, figure 6.5B display the graph that corresponds to patterns shown in Figure 6.5A. Internally, Sparrow's patterns are represented by a special type of node called *pattern node*. These nodes implement the different synchronizations operators supported by Sparrow (see section 4.3). Additionally, each pattern node maintains a history (called the *buffer*) of previously matched messages.

Figure 6.4 shows a simplified view of the internal representation of a Sparrow actor, and how messages in that actor's Elixir-level inbox are transferred to the pattern engine's *virtual inbox*. The matching process starts by transferring each received message from the actor's Elixir-level inbox into its virtual inbox (see `Step 1`). Later JuPITer tries to match each new message against a group of patterns for which it is relevant; we call this process a *match-cycle*. The discrimination network's *root node* serves as the entry point of new messages for the matching process. This node will determine potential pattern nodes based on the message's type and will forward it to them. After this step, the message will "flow through" the discrimination network until a pattern node is found with a successful match for its conditions (see `Step 2`). In that case, the message is *consumed* by its reaction(s), which will be sequentially executed by the actor's *reaction manager* (see `Step 3`). Otherwise, the message remains in the node's buffer until a successful match is completed or until the message expires. The above process is an essential tool for the engine's incremental matching strategy. This strategy is based on a custom implementation of the RETE algorithm [22] that will be explained in section 6.3.1. Furthermore, it implements a *single pattern selection* and *selected message consumption* policies [89]. The former guarantees that a pattern matches *at most once* per match-cycle. The latter guarantees that a pattern can consume a message *only once*. However, multiple patterns can consume the same

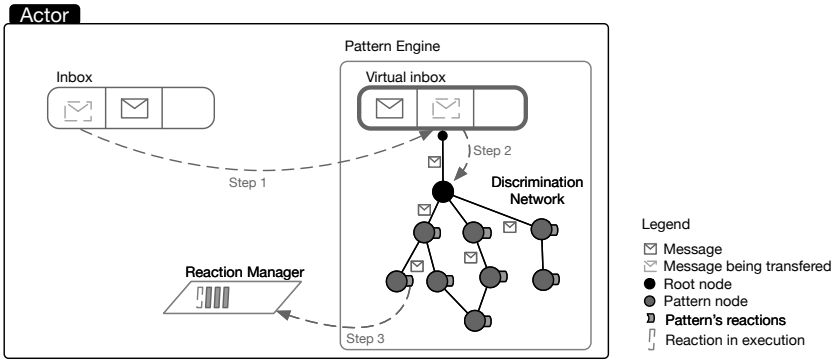


Figure 6.4 Overview of the internal representation of a Sparrow actor

message (only once). Although in this figure patterns are represented by a single type of node, several subtypes exist, each of them addressing a particular type of pattern defined in chapter 4. In the next section, we detail the matching algorithm of JuPITer.

6.3.1 A RETE-based Matching Algorithm

As we mentioned in the previous section, JuPITer's matching mechanism is inspired by the RETE [22] algorithm. This algorithm is a well-established efficient pattern-matching algorithm for implementing production rule-based systems. RETE maintains a network of nodes through which facts (a.k.a. events or messages) are filtered. In the rest of this chapter, we will use the term *message* instead of *fact* to refer to the entry values of the RETE algorithm. RETE avoids re-evaluating the conditions of its rules each time a new message arrives. This incremental matching strategy is supported by storing data in-between match cycles. From its original definition, we can distil three main types of nodes:

- *Alpha nodes* are responsible to filter individual messages based on simple conditional tests which match messages attributes against constant values.
- *Beta nodes* are responsible to perform joins between different messages. Unlike alpha nodes, they consist of two-input nodes.
- *Terminal nodes* are responsible to execute the body of a rule.

RETE has been the subject of multiple extensions such as [39, 54]. In this dissertation, we present an adaptation of this algorithm to support the message synchronization abstractions provided by Sparrow. Briefly, JuPITer's matching algorithm extends RETE in four aspects:

- Advanced filters mechanisms to filter individual messages and group of messages by their content and time constraints.
- Flexible matching selection policy.
- Explicit support for conjunctions, and disjunctions of messages.
- Detection of the absence of messages in a time window.

We now explain in detail the different type of RETE network nodes that realize this behaviour:

Root Node serves as the entry point for new messages into JuPITer's node network. It will determine potential alpha nodes based on the message's type and will forward the message to these alpha nodes.

Alpha Nodes match a full *pattern selector* of a *named pattern* (see section 4.3), instead its individual attributes (as in the original RETE). This decision was inspired by the need to reuse and compose patterns. By having this level of filter granularity, the hierarchy of nodes can be easily established. Messages that not satisfy the filter conditions of an alpha node are immediately discarded and garbage collected. In contrast, successfully matched messages are forward to its list of child omega nodes and its terminal node.

Opaque Alpha Nodes are a special type of alpha node generated by *anonymous* patterns (see section 4.3). Unlike regular alpha nodes, they cannot be reused by other patterns.

Omega Nodes extend the filter conditions of alpha nodes and are responsible to implement the computational logic of the different pattern operators such as *time window*, *extensional sequencing*, etc (see section 4.3). The parent of an omega node can be any other type of node except a terminal node.

A

```
1  defmodule LightManager do
2   use Sparrow.Actor
3
4   pattern motion as {:motion, id, :on, room}
5   pattern light as {:light, id, status, room}
6
7   pattern on_motion as motion
8                     and light{status= :off}
9                     and {:amb_light, id, illuminance, room},
10                    options: [last: true]
11                    when illuminance < 40
12
13  pattern no_motion as not motion[window: {2, :mins}]
14                    and light{status= :on},
15                    options: [last: true]
16
17  reaction turn_on_light(l, i, t), do:  # send on command
18  reaction turn_off_light(l, i, t), do: # send off command
19
20  react_to on_motion, with: turn_on_light
21  react_to no_motion, with: turn_off_light
22
23 end
```

B

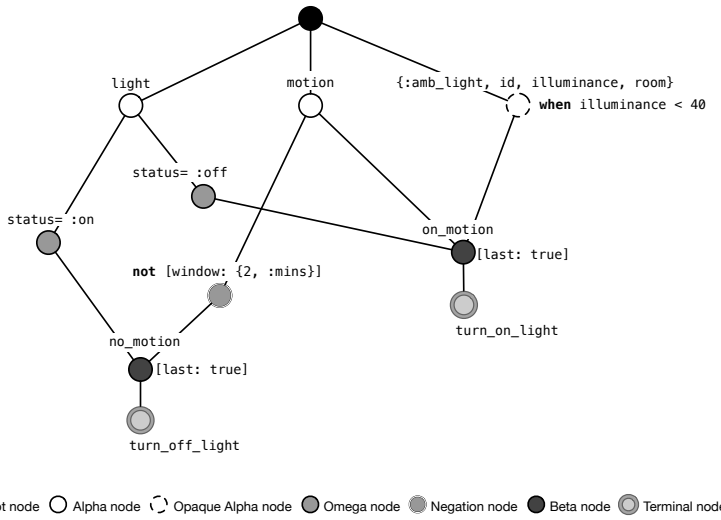


Figure 6.5 Internal representation of messages patterns in Sparrow

Negation Nodes are a special type of omega node that will be activated if after an *initial* match no other message matches in a particular time window. Whenever a new message matches, its internal timer is reset.

Beta Nodes are responsible to perform joins between different messages. Unlike traditional RETE beta nodes, they may consist of more than two input nodes.

Terminal Nodes are responsible to request the execution of one or more reactions attached to a pattern. These nodes differ from the traditional RETE terminal nodes in two aspects. First, a JuPITer's terminal node may contain one or more reactions. Second, they do not execute the body of a reaction, instead, they schedule its execution. For this reason, unlike the aforementioned nodes, terminal nodes are merged with their respective node (i.e., alpha, opaque alpha, omega, or beta). However, to facilitate the understanding of the actor's pattern engine we visually represented it as a standalone node (see figure 6.5). The reactions of a pattern or multiple patterns are scheduled since they can modify the state of its actor. As we explained early in this section, reactions are executed one at a time by the actor's reaction manager. The reaction manager guarantees that a reaction is always executed with the latest state of the actor.

Figure 6.5 illustrates how message patterns of a Sparrow's actor are represented in its embedded JuPITer engine. At the top (A), observe a variant of listing 4.2 introduced in section 4.2, which declares two composite patterns (lines 7-11, 13-15) and two reactions (lines 17-8) to turn on/off the light of any room. The only change in this variant is that the pattern to detect low ambient light has been defined as an anonymous pattern (line 9). At the bottom (B) of this figure, observe the generated pattern engine for the actor `LightManager` and its patterns. For example, two *alpha nodes* and one *opaque alpha node* are responsible to filter incoming messages based on their values. As *light* and *motion* were declared as *named* patterns we used their names in the graph. In contrast, for the anonymous pattern that allows us to detect low ambient light, we used the full definition of its selector and guard. This example also showcases the use of two omega nodes to filter the status (on/off) of light messages, and a *negated omega node* to detect the absence of movement in a room for a time window of two minutes. Furthermore, two *beta nodes* are used to represent both composite patterns (i.e., `no_motion`, `on_motion`). Finally, observe each beta node has a terminal node with a reaction attached to it. This corresponds to the fact that we only registered a reaction to each of these nodes.

Technically in Elixir, JuPITer's node network is implemented as a sub-network of actors. Each node is represented by a particular type of actor that was designed to execute a specific task. The state of the network is currently implemented *in-memory* and its nodes form part of an actor supervision tree [77]. JuPITer has a specialized actor, called supervisor with one purpose: monitoring JuPITer's node network. Although this supervisor provides fault-tolerance to the actor's engine, in its current implementation (i.e., *in-memory*) if a node crashes, its data (and that of its children) will be lost since that part of the network must be rebuilt. This limitation can be fixed in future versions of JuPITer by adding a state persistence mechanism for its network. For example, we might use a `@persistent` annotation for Sparrow's actors.

6.4 Tool Support

New programming abstractions designed to facilitate the coordination of actors are beneficial (see chapter 7). However, if they lack support for features typically found at the IDE level (such as syntax highlighting, autocomplete, and debugging) their adoption by software developers will be limited. In the next two sections, we describe two basic tools that aim to facilitate the development of Sparrow-based programs. We will use the Sparrow actor defined in listing 6.6 to illustrate the language tooling support described in this section. The `CourtLightsDemo` actor illustrate a typical actor with several types of Sparrow's patterns.

6.4.1 Visual Studio Code Extension

Sparrow as a DSL benefits of the existing tools (e.g., debugger[5], observer[6]) provided by its host language, Elixir. However, its custom syntax and macros are obviously not supported by Elixir's extensions for standard integrated development environments (e.g., IntelliJ IDEA) and code editors (e.g., VS Code). To overcome this limitation, we have extended an existing Elixir extension for VS Code, called *vscode-elixir-ls*[7]. More specifically, we extended its implementation of Microsoft's IDE-agnostic Language Server Protocol and VS Code debug protocol to add support for Sparrow's syntax

[5] `https://elixir-lang.org/getting-started/debugging.html#debugger`
[6] `https://elixir-lang.org/getting-started/debugging.html#observer`
[7] `https://github.com/elixir-lsp/vscode-elixir-ls`

Listing 6.6 Example of a complex actor in Sparrow

```
defmodule CourtLightsDemo do
  use Sparrow.Actor

  pattern motionSensor as {:motion, zoneId}
  pattern rainSensor as {:rain_sensor, zoneId, visibility}

  pattern courtLights as {:court_lights, zoneId, status}

  pattern noMotion as not motionSensor[window: {30, :secs}]

  pattern noiseSensor as {:noise, zoneId, noise}[group: 10]


  pattern courtLightsOff as courtLights{ status= :OFF }
  pattern courtLightsOn as courtLights{ status= :ON }

  pattern turnCLightsOn as motionSensor
                   and rainSensor{visibility < 30}
                   and courtLightsOff
                   or noiseSensor{noise> 90}

  reaction reaction1(_), do: # code
  reaction reaction2(_), do: # code
  reaction reaction3(_), do: # code

  react_to rainSensor, with: reaction1
  react_to noMotion, with: reaction2
  react_to turnCLightsOn, with: reaction3
end
```

and macros. We call our VS Code extension *Sparrow VS* and its language server protocol implementation *Sparrow LS*. Both source code repositories can be found online[8][9].

Figure 6.6 shows an example of the suggested code autocompletation for the *macro pattern*. Similarly, developers will be assisted with all other Sparrow's macros as well. Sparrow VS also supports automatic and

[8]Sparrow VS Code Extension - `https://github.com/rhumbertgz/sparrow-vs`
[9]Sparrow Language Server Protocol - `https://github.com/rhumbertgz/sparrow-ls`

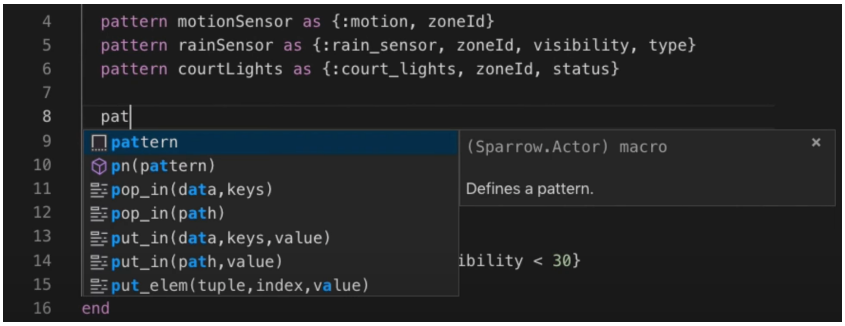


Figure 6.6 Autocomplete support for Sparrow abstractions

incremental Dialyzer analysis [2] of Sparrow programs. Using this built-in static analysis tool inherited from the Erlang/Elixir ecosystem, Sparrow VS provides inline reporting of build warnings and errors. For example, figure 6.7 shows an example of an inline build error while trying to extend a non declared `motion` pattern. As we observe in line 4 of listing 6.6 the right pattern name is `motionSensor`.

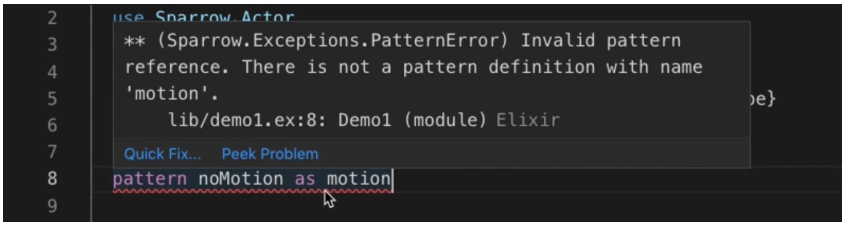


Figure 6.7 Autocomplete support by the Sparrow VS Code extension

Figure 6.8 shows another example of an inline build error that is caused by the use of a non-supported operator (`windows`). In this case, the right operator is `window`. The interested reader can see a full demo of Sparrow VS on YouTube[10].

6.4.2 Real-time Monitoring Tool

Sparrow VS adds language support for our DSL abstractions in a modern code editor. However, this does not satisfy to debug complex Sparrow's

[10]Sparrow VS Demo - `https://youtu.be/IyCN7nolv6I`

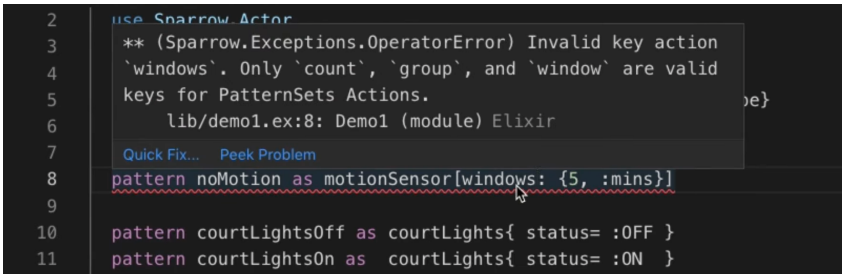


Figure 6.8 Incremental compilation output with an inline build error

actors. Developers need to have an understanding of how an actor is internally represented and how its embedded pattern engine works. To provide developers with such information about a running Sparrow program, we have developed a real-time monitoring tool, called *JuPITer monitor*. This tool allows developers to inspect the current state of an actor's virtual-inbox and pattern engine. This tool is meant to be used during the development of an application, and it is fed by events generated by each h of the main activities undertaken by a running Sparrow-based system: create new actors, send messages, and react to pattern matches. We have integrated the JuPITer monitor as a *mix*[11] task. Developers can easily monitor an application by executing the command `mix spw.run -monitor` in the root directory of that application. The interested reader can watch a full demo of the JuPITer monitoring tool on YouTube[12]. During the initialization of the actors (and its patterns/reactions) of a monitored application, events describing that process will be generated and visualized by the web interface of this tool. Such events are sent to the monitoring application using the macro `notify` defined by the module `AppMonitor`. We decided to implement `notify` as a macro instead of a regular function to optimize the Sparrow's code for applications ready to run on production. In production, `AppMonitor.notify` expressions would be completely removed from the AST of Sparrow's core libraries. We can see the resulting Sparrow optimized version as a minified or compressed version of a JavaScript library (e.g., `jquery.min`). This optimization was inspired by Elixir's Logger library that in a similar way skips log calls.

[11]Mix is a build tool that ships with Elixir that provides tasks for creating, compiling, testing, managing dependencies, etc.

[12]JuPITer monitor demo - `https://youtu.be/pl-I9qh3ZgU`

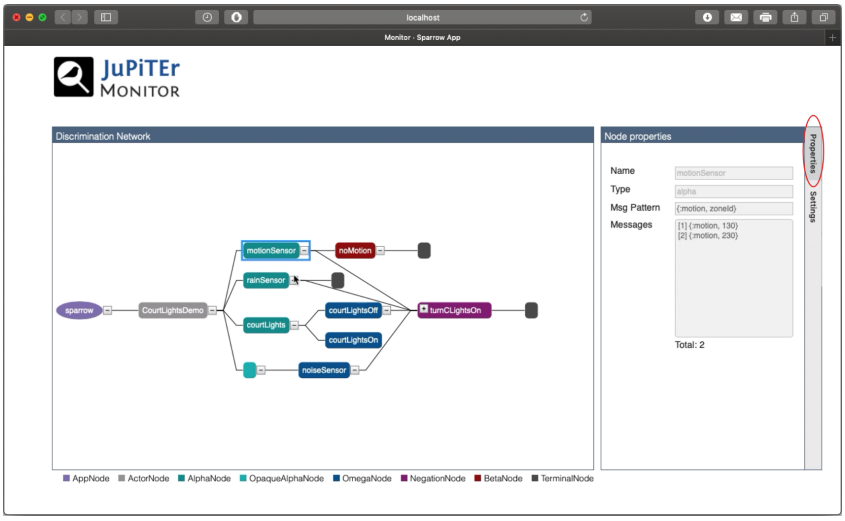


Figure 6.9 Internal representation of Sparrow's patterns in JuPITer

Figure 6.9 shows the visual representation of the pattern engine that sits in the actor created by the code shown in listing 6.6 on page 112. Here we observe how developers can select a node (e.g., `motionSensor`) and visualize, in real-time, its state (e.g., name, type, selector, messages) in the *Properties* tab. Developers can also customize the visual representation of the engine's nodes network (*Settings tab*) to help them to understand the composition of complex patterns. For example, in figure 6.10 we enabled the overview panel to facilitate the navigation of large nodes network. Furthermore, we enabled the highlighting of the links of connected nodes. As we observe, by selecting the node `CourtLightsDemo` its input and output links are highlighted.

6.5 Conclusion

The implementation of Sparrow's primitives and auxiliaries functions heavily rely on Elixir's pattern matching mechanism. Sparrow's patterns inherit Elixir's pattern matching capabilities to simplify the definition of message filter constraints.

The limitations of Elixir's macro system restricted the expressiveness of Sparrow's coordination abstractions.

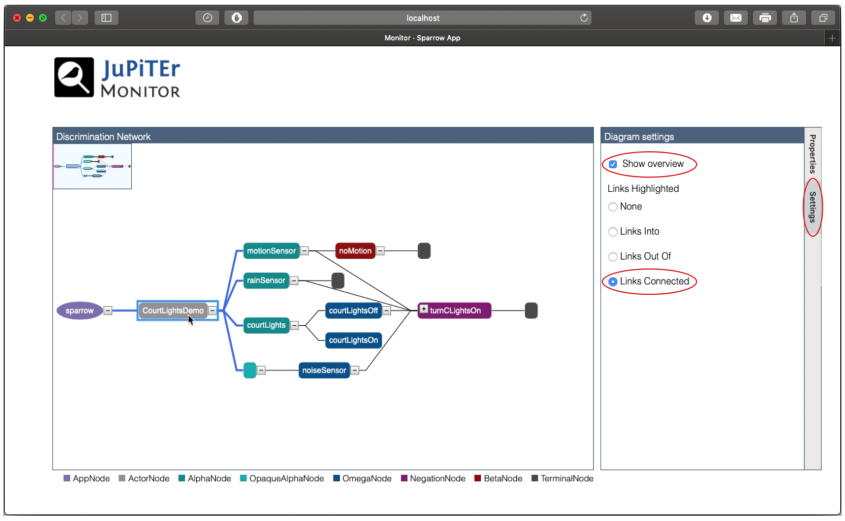


Figure 6.10 Visual properties for the discrimination network representation

The custom implementation of the RETE algorithm used by the embedded pattern engine of a Sparrow actor helped to support the advanced coordination operators imported from CEP systems. This implementation could be still improved in future versions, e.g., by adding persistence support of its nodes network.

The development of a VS code extension and a real-time monitor tool for Sparrow-based programs proved advantageous during the implementation and debugging of Sparrow's abstractions. Furthermore, these tools could also facilitate its adoption and extension by other researchers in the future.

CHAPTER 7

Validation

In this chapter, we evaluate the expressiveness of Sparrow by means of its solutions for the seven smart home scenarios formulated in chapter 2. Section 7.1 does an initial comparison of the solutions in openHAB, Hass, Elixir, and Sparrow for *one* of these scenarios. Section 7.2 presents a more detailed quantitative evaluation approach. This approach is based on an analysis of the lines of code required by the solutions of the above technologies to express the coordination logic of *all* these smart home scenarios.

7.1 A Code Comparison Analysis of Scenario 5

In this section, we compare three implementations of scenario 5 in openHAB, Elixir, and Sparrow. We chose scenario 5 because it exemplifies a set of **coding concerns** that developers have to take care of manually. These coding concerns are:

- **State management** is the code that is used to save temporal data required by the ongoing coordination process.
- **Windowing management** highlights the code needed to discard messages that do not satisfy the pattern's timing constraints.

- **Sequencing control** is the code is needed to enforce a particular message order.

- **Pattern definition** emphasizes the code used to express the type of messages to be synchronized and their content-based conditions.

Regarding the programming alternatives, we obviously chose Elixir because it is the host language of our DSL and it provides synchronization abstractions similar to other mainstream actor languages (e.g., Erlang, Scala). Due to the similarity between the solutions of openHAB (Jython) and Hass (Python), we omitted the latter. This decision was also motivated by the fact that the openHAB community was the most responsive and helpful during the validation of the scenarios formulated in chapter 2. However, the interested reader can find the implementation of all the smart home scenarios in our GitHub repository [56].

We observe in figure 7.1 that for the scenario 5, Sparrow's solution requires the least amount of code. But more importantly, Sparrow manages to completely hide or significantly reduce these coding concerns from the developer. For example, in the implementations of both openHAB (lines 5-7, 12-13, 18-19, 30-31) and Elixir (lines 4-5, 13, 21, 24, 27), we had to manually track each sensor's most recent message (*state management*). However, Sparrow automatically keeps track of message timestamps and therefore it can mostly hide this concern from the developers. Furthermore, our DSL (lines 10, 13) provides a compact syntax to discard messages based on time windows (*windowing management*) instead of forcing developers to write complex and nested `if` expressions to carry out that action as is the case in openHAB (lines 21-22, 33-34) and Elixir (lines 8, 16). Similarly, Sparrow's sequencing operator (lines 10, 13) offers a short syntax to specify the desired matching order (*sequencing control*). In contrast, such synchronization operation in openHAB (lines 21, 24, 33, 36) and Elixir (lines 8-9, 16-17) is directly proportional to the number of different messages involved. Finally, although the last concern (*pattern definition*) is the most crucial for the coordination process, both openHAB (lines 8-9, 15-16, 27-28) and Elixir (lines 7, 15, 23) abstractions only define when to react to single messages.

In brief, these implementations of scenario 5 give a strong indication that Sparrow's abstractions succeed to provide more expressive and compact solutions for coordinating a group of actors. The inability to express the

A

```
from core.rules import rule
from core.triggers import when
from java.time import ZonedDateTime as ZDT

c_door = ZDT.now().minusHours(24)
m_hall = ZDT.now().minusHours(24)
m_door = ZDT.now().minusHours(24)

@rule("(Py) Front Door Opened")
@when("Item Front_Door_Contact changed to OPEN")
def front_door_opened(event):
    global c_door
    c_door = ZDT.now()

@rule("(Py) Motion Detected - Entrance Hall")
@when("Item Entrance_Hall_Motion changed to ON")
def entrance_hall_motion(event):
    global m_hall, m_door, c_door
    m_hall = ZDT.now()

    if m_door.isBefore(m_hall.minusSeconds(60)):
        return

    if m_hall.isAfter(c_door) and c_door.isAfter(m_door):
        # code logic for arriving home

@rule("(Py) Motion Detected - Front Door")
@when("Item Front_Door_Motion changed to ON")
def front_door_motion(event):
    global m_hall, m_door, c_door
    m_door = ZDT.now()

    if m_hall.isBefore(m_door.minusSeconds(60)):
        return

    if m_door.isAfter(c_door) and c_door.isAfter(m_hall):
        # code logic for leaving home
```

B

```
defmodule Automation5 do
  import Timex

  def loop({m_door, m_hall, c_door}) do
    state =
      receive do
        {:motion, _id, :on, :front_door, m_door_dt} ->
          if before?(shift(m_door_dt, seconds: -60), m_hall)  do
            if after?(m_door_dt, c_door) and after?(c_door, m_hall) do
              # code logic for leaving home
            end
          end
          {m_door_dt, m_hall, c_door}

        {:motion, _id, :on, :entrance_hall, m_hall_dt} ->
          if before?(shift(m_hall_dt, seconds: -60), m_door)  do
            if after?(m_hall_dt, c_door) and after?(c_door, m_door) do
              # code logic for arriving home
            end
          end
          {m_door, m_hall_dt, c_door}

        {:contact, _id, :open, :front_door, dt} ->
          {m_door, m_hall, dt}
      end

    loop(state)
  end
end
```

C

```
defmodule Automation5 do
  use Sparrow.Actor

  pattern motion as {:motion, id, :on, location}
  pattern m_front_door as motion{location= :front_door}
  pattern m_entrance_hall as motion{location= :entrance_hall, id~> mid}
  pattern c_front_door as {:contact, cid, :open, :front_door}

  pattern occupied_home as m_front_door and c_front_door and m_entrance_hall,
                           options: [ interval: {60, :secs}, seq: true, last: true ]

  pattern empty_home as m_entrance_hall and c_front_door and m_front_door,
                        options: [ interval: {60, :secs}, seq: true, last: true ]

  reaction activate_home_scene(l, i, t), do: # code logic for arriving home
  reaction activate_leave_scene(l, i, t), do:  # code logic for leaving home

  react_to occupied_home, with: activate_home_scene
  react_to empty_home, with: activate_leave_scene
end
```

State management | Pattern definition | Windowing management | Sequencing control

Figure 7.1 Solution for scenario 5 in openHAB (A), Elixir (B), and Sparrow (C)

whole coordination process using declarative patterns forces developers to shift their focus from *when* to react to *how* to do it. In contrast, in Sparrow (see figure 7.1.C), a developer focuses on the declaration of patterns, and lets the run-time figure out how to match its constituents.

Table 7.1 Total LoC of the different solutions for the seven smart home scenarios

	openHAB	Hass	Elixir	Sparrow
State management	29	31	55	0
Sequencing control	2	2	2	2
Windowing management	20	15	14	7
Pattern definition	24	20	31	16
Total lines of code (LoC)	75	68	102	25

7.2 Quantitative Evaluation

In this section, we do a preliminary quantitative evaluation of **all** the scenarios solutions in openHAB, Hass, Elixir, and Sparrow. To get a fair comparison between our actor-based solutions and the ones from the smart home platforms, we published these solutions on both community's forums (see openHAB [61] and Hass [57] topics). This happened *during* the development of Sparrow. The publication of our solutions allowed us to get feedback and incrementally arrive at a solution that could be used by experts of these communities.

Table 7.1 shows the sum of the total LoC required by the different implementations for each of the coding concerns identified in section 7.1. For a detailed LoC breakdown of each solution we refer the reader to appendix A.1. Like in our analysis for the solutions of scenario 5, we do not consider code not related to the coordination process (e.g., imports, reaction logic of the automations), and we limit all lines to a maximum of 95 characters. Next, we manually tagged each LoC related to the identified concerns to obtain the relevant code for our analysis. These tags were later retrieved and aggregated in a CSV file (a *dataset* per concern) using a helper script (`statistics.exs`) located in the root directory of each solution group (e.g., openHAB, Sparrow). **In this table, we observe that Sparrow required less LoC for each coding concern in all its solutions than the other implementations, except for the sequencing control concern where all have the same value.** However, as we mentioned earlier in section 7.1, for this concern Sparrow is the only one for which the

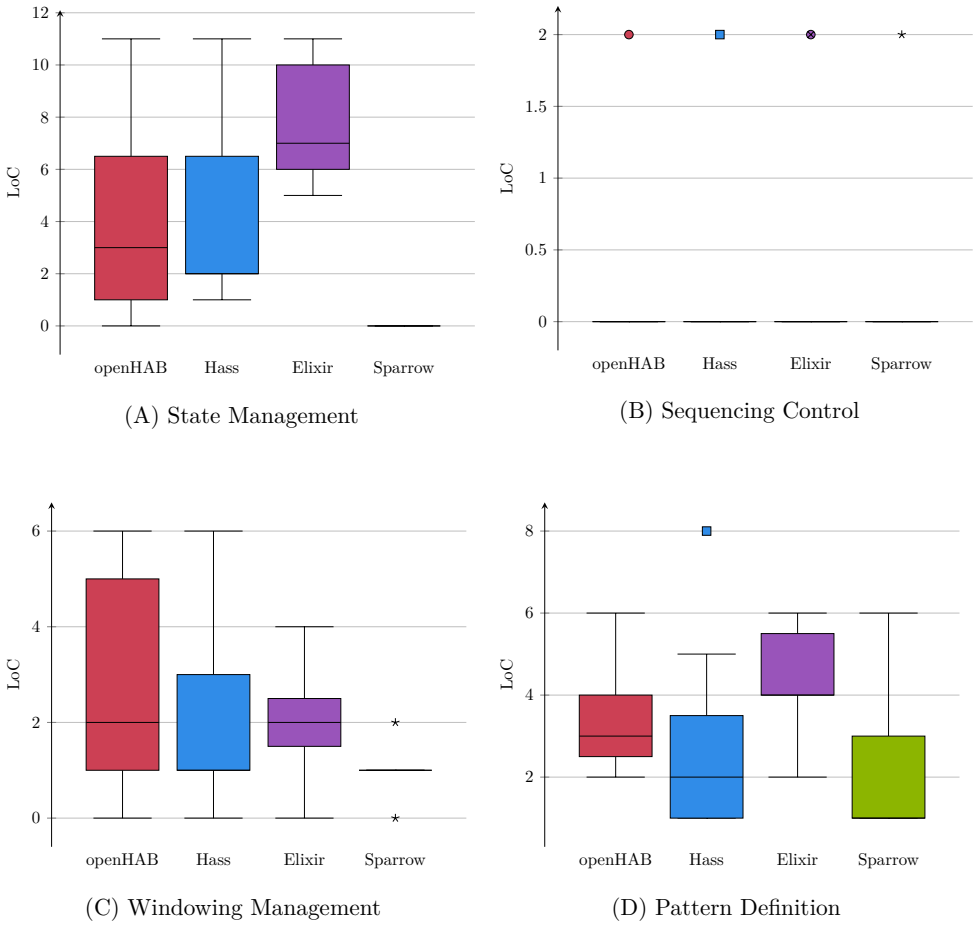
Table 7.2 Statistical overview of the solutions per coding concern

	Mean	Std	Min	25%	50%	75%	Max
State management							
openHAB	4.14	4.26	0.00	1.00	3.00	6.50	11.00
Hass	4.43	4.20	1.00	2.00	2.00	6.50	11.00
Elixir	7.86	2.41	5.00	6.00	7.00	10.00	11.00
Sparrow	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Sequencing control							
openHAB	0.29	0.76	0.00	0.00	0.00	0.00	2.00
Hass	0.29	0.76	0.00	0.00	0.00	0.00	2.00
Elixir	0.29	0.76	0.00	0.00	0.00	0.00	2.00
Sparrow	0.29	0.76	0.00	0.00	0.00	0.00	2.00
Windowing management							
openHAB	2.86	2.48	0.00	1.00	2.00	5.00	6.00
Hass	2.14	2.12	0.00	1.00	1.00	3.00	6.00
Elixir	2.00	1.29	0.00	1.50	2.00	2.50	4.00
Sparrow	1.00	0.58	0.00	1.00	1.00	1.00	2.00
Pattern definition							
openHAB	3.43	1.40	2.00	2.50	3.00	4.00	6.00
Hass	2.86	2.67	1.00	1.00	2.00	3.50	8.00
Elixir	4.43	1.40	2.00	4.00	4.00	5.50	6.00
Sparrow	2.29	1.98	1.00	1.00	1.00	3.00	6.00

LoC of its solutions does not grow depending on the number of messages being synchronized.

Besides the results shown in table 7.1, we have applied a descriptive statistic analysis over the above datasets in order to compare Sparrow beyond simple LoC counting. Table 7.2 summarizes the central tendency, dispersion and distribution of the required LoC for each coding concern.

In order to facilitate the analysis of the results shown in table 7.2, we use a boxplot visualization (see figure 7.2). The source code of this analysis and its datasets are publicly available in [63].



(A) State Management

(B) Sequencing Control

(C) Windowing Management

(D) Pattern Definition

■ **Figure 7.2** Expressiveness of the solutions per coding concern and platform

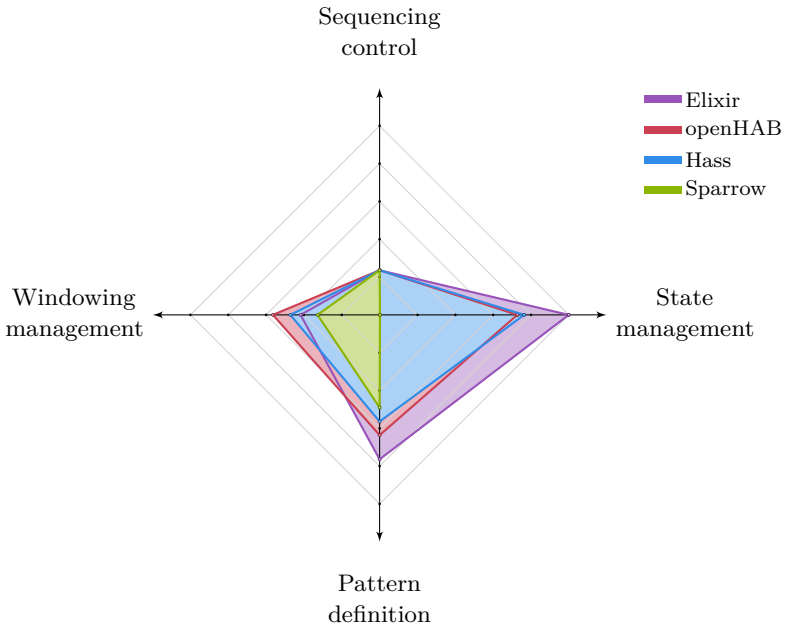
Figure 7.2.A shows the distribution of the LoC required to handle the *state management* for the seven smart home scenarios introduced in chapter 2. Since Sparrow does not require the developers to manually track the state relevant for the coordination process in its solutions, we only focus on the interpretation of the results of openHAB, Hass, and Elixir. First, we observe that in general, both smart home platforms solutions

require less LoC than the Elixir ones. However, 50% of Elixir's solutions have a lower LoC variability (*less is better*) than the solutions of openHAB and Hass. For example, openHAB has the highest LoC variability, with 5.5 lines as their interquartile range (IQR). Second, their solutions do not have a normal LoC (data) distribution since their boxplot representations are upper-skewed. Nevertheless, the third quartile (75%) of openHAB and Hass solutions has a more LoC variability than the third quartile of Elixir. Briefly, although Elixir's solutions require more LoC, they have less LoC variability between them leading to more uniform solutions.

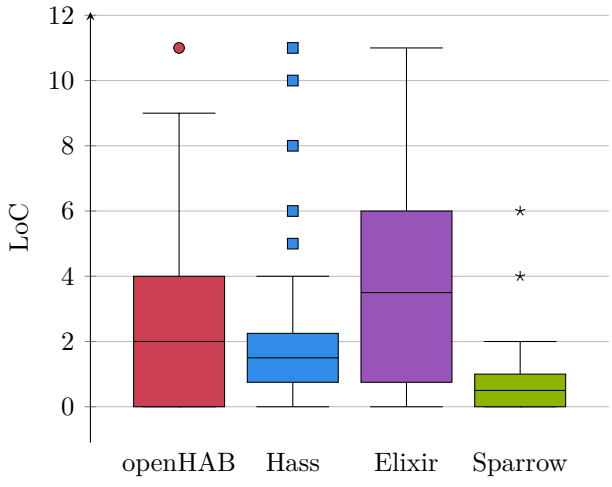
Figure 7.2.B shows that all solutions have the same LoC distribution for *sequencing control*. We attribute this result to the fact that only one scenario required a particular sequence of messages, and the number of involved messages was relatively small (only three). Despite these results, the required LoC in openHAB, Hass, and Elixir will grow based on the number of messages being synchronized. Furthermore, the complexity of that code will also increase since it will be mostly based on `if` statements. In contrast, Sparrow solutions will require the same expression `[seq: true]` in order to enforce a sequence order for more than two messages.

Despite the outliers of Sparrow's solutions shown in figure 7.2.C, it still provides the most expressive solutions. We observe that as Sparrow's inner quartiles overlap they have less LoC variability. This means that 75% of Sparrow's solutions are at minimum 2x and a maximum 5x more compact than the solutions from openHAB, Hass, and Elixir. Although Hass and Sparrow share the same median value (1 LoC) in their solutions, the solutions of our DSL followed by the Elixir ones have less LoC variability than the rest. **Briefly, the solutions of both actor-based languages are more consistent concerning the required LoC between their solutions than the one from the smart home platforms.**

Finally, figure 7.2.D shows that Sparrow offers the most consistent solutions. As we observe in this figure, Sparrow's first and second quartiles (median 50%) overlap, so the 50% of the cases Sparrow requires the lowest number of LoC of all implementations (e.g., openHAB). Additionally, Sparrow's third quartile is also smaller than the rest, as result the 75% of the cases its solutions are more compact than the others. Although Sparrow falls behind openHAB to offer less LoC variability in its IQR, this can be justified by the fact that Sparrow's developers focus more



(A) Overall results per coding concerns



(B) Overall Results

Figure 7.3 Summary of the solutions for the seven scenarios

on the definition of patterns than any other of the concerns identified in section 7.1.

Figure 7.2 shows that Sparrow seems to succeed in reducing the effects of those coding concerns above the synchronization of messages. Figure 7.3 summarizes our analysis using two different types of visualization. The first one, figure 7.3.A uses radar chart to plot the data presented in table 7.1. In this particular graph, values closer to the center represent less LoC required and, as a result, more expressiveness in the code. By representing each coding concern as an axis in this graphical method, we can visually observe how Sparrow's abstractions have greater expressiveness along all axis, except in sequencing control where all implementations have the same value. The second one, figure 7.3.B illustrates that Sparrow's solutions are the most compact of all implementations, followed by the solutions of Hass. In general, the 75% of the solutions our DSL required less LoC than the 50% of the other solutions. At the same time, we observe that Elixir's solutions require more effort from their developers. The data used in this last box plot corresponds to the combination of the individual data frames used in figure 7.2.

7.3 Conclusion

The implementation of the seven scenarios described in chapter 2 allowed us to showcase the expressiveness of Sparrow. At the same time, our empirical and preliminary evaluation allowed us to identify four coding concerns faced by developers during the coordination of a group of smart devices. Finally, the comparison between the solutions using openHAB, HAS, Elixir, and Sparrow shows that our DSL was more effective at reducing the amount of code needed to express the intended synchronization behavior. However, a more rigorous evaluation must be conducted in the future to complement this preliminary judgement.

CHAPTER 8

Conclusion

In this final chapter, we reflect on the requirements for advanced message synchronization abstractions formulated and motivated in chapter 2, and we highlight how the Sparrow DSL addresses them. We discuss the current limitations of Sparrow and give an overview of future research directions.

8.1 Summary and Contributions

Since its invention in 1973, the actor model has been the subject of several extensions to facilitate the coordination of a group of actors. Researchers have explored different approaches from *centralized* to *local* coordination using several techniques based on *reflection*, *session types*, *constraint-handling rules*, and *join patterns* just to mention a few. In this dissertation, we identified a suite of common message synchronization requirements using seven smart home scenarios for which, up to now, have no or very limited support exists in modern actor-based languages. The relevance of these requirements was confirmed by more than 700 developers in the smart home community. We summarize these requirements as:

1. Advanced message filter abstractions to support timing constraints and negation.

2. A flexible message selection policy that allows an actor to match a particular message instead of just the oldest message in the actor's inbox.

3. Advanced message matching mechanism to support matching of multiple messages with or without a particular order.

4. Advanced message matching mechanism to accumulate and match a quantified or unquantified number of messages.

5. Support for further transformation and filter of matched messages before the corresponding reaction is actually executed.

Sparrow's patterns form a research contribution in the context of actor model extensions. They are motivated by the above message synchronization requirements. As a result, the actor's interface is empowered with advanced declarative message matching capabilities. In this dissertation, we validate the expressiveness of Sparrow's patterns by comparing its solutions for the seven smart home scenarios against the ones of two popular smart home platforms and a modern actor-based language. The choice of the technologies used in our validation had two main motivations. First, they are state-of-the-art and popular technologies within their communities. Second, they provide a level of synchronization abstractions similar to the ones used in their alternative technologies. We now revisit the contributions outlined in this dissertation.

A suite of common synchronization requirements. Our first contribution is the formulation of a suite of common synchronization requirements needed to coordinate modern actor-based systems. These requirements were presented in chapter 2. We started by developing seven smart home scenarios inspired by real automation rules found on smart home community forums. We chose this particular type of cyber-physical systems because they exhibit a wide range of synchronization requirements. From these smart home scenarios, we designed and ran an online poll to validate that each of them was a representative example of the different synchronization needs found in the wild. Based on the results of our poll we concluded that our scenarios are good examples of concerns that live within the smart home community. Later we distilled five categories of synchronization requirements needed to express the seven smart home scenarios. During the implementation of these scenarios, we found that two popular smart

home platforms (i.e., openHAB, Hass) and a modern actor-based language (i.e., Elixir) did not have good support for the identified synchronizations requirements.

A survey of existing actor-based coordination approaches and CEP operators. Our second contribution is an extensive survey of related actor coordination proposals and CEP operators presented in chapter 3. Our literature review analyzed a large list of contributions in these two fields and compared their support for the synchronization requirements identified in chapter 2. First, we started by reviewing the literature on extensions to the point-to-point communication model of the traditional actor model. Second, we studied approaches based on reflection and runtime verification used to extend the synchronization abstractions supported by the actor model. Third, we scrutinised more local approaches which seamlessly integrated their synchronization abstractions into an actor-based language. Since we found the approaches in the above groups had limited support for the synchronization requirements identified in chapter 2, we drew inspiration from the existing types of synchronization mechanisms supported by CEP systems.

A domain-specific language for advanced coordination of heterogeneous actors. Our third contribution is the design and implementation of the Sparrow DSL (see chapter 4). Sparrow's coordination abstractions target the synchronization requirements identified in chapter 2. These coordination abstractions were implemented as *advanced messages patterns* that can be reused and composed. Sparrow's patterns use pattern-matching techniques and a set of operators to filter, accumulate and transform messages. These patterns were built as Elixir macros, of which the output code is optimized and validated during the compilation of a Sparrow-based program. Furthermore, they can seamlessly integrate into standard Elixir applications and can fully leverage Elixir's primitives and libraries. The validation of our DSL demonstrated the expressiveness of its coordination abstractions. Furthermore, it also demonstrated that Sparrow's solutions were more effective to reduce the effects of non-functional concerns on the synchronization of messages than the other compared technologies. To the best of our knowledge, Sparrow is the first actor-based DSL that combines join patterns and complex event processing techniques into one coherent model.

A formal calculus of Sparrow called NEST and its mechanization. Our fourth contribution is the definition and implementation of a mechanized formal calculus to precisely describe the semantics of the core coordination abstractions of Sparrow (see chapter 5). We called this calculus NEST. NEST was implemented in Redex [21]. Using Redex's tests suite, we exhaustively tested NEST's syntax and reduction rules. However, we could not define randomized tests for NEST's patterns reduction rules. These reduction rules require specific randomized messages sequences and such requirements could not be expressed with the built-int Redex's test suite. To overcome this limitation, we implemented *a randomized pattern test generation algorithm*, and two new functions to *test* (`pattern-test`) and *trace* (`pattern-traces`) the correct resolution of NEST's reduction rules for patterns. Our algorithm enabled us to verify not only whether valid messages sequences are detected by the calculus, but also whether invalid messages sequences are not accidentally accepted. We constructed this method to validate our semantics without having to construct formal proofs.

A novel RETE-based matching algorithm. Our fifth contribution is a custom implementation of the RETE [22] algorithm to support an incremental matching mechanism of Sparrow's patterns (see section 6.3). This implementation is used by the embedded message pattern engine of Sparrow's actors, called JuPITer. Our extension adopts a higher level of granularity, but more advanced filter conditions than the original RETE to facilitate the reuse and composition of patterns. It also introduces new types of nodes into JuPITer's nodes network to support the synchronization abstractions implemented by Sparrow. A distinct characteristic of our RETE-based implementation is that it only processes messages from a single actor, instead of the common approach of using a global RETE implementation for a whole system. Each Sparrow's actor has its own RETE-based JuPITer instance.

Basic Tools Support. Our sixth contribution is the implementation of two software tools to support the development of Sparrow-based programs (see section 6.4). On the one hand, we extended the syntax and language server protocol of an existent Elixir VS Code extension with a double purpose. First, add support for Sparrow's syntax highlighting and macro autocompletion. Second, provide inline reporting of build warnings and

errors of Sparrow programs, using an automatic and incremental static analysis based on Erlang's Dialyzer tool. On the other hand, we complemented the above VS Code extension with a real-time monitor tool to allow developers to inspect running JuPITer's instances of a Sparrow program. The development of both tools had a significant impact during the implementation and debugging of Sparrow's abstractions. Furthermore, they also can facilitate the adoption and extension of Sparrow by other researchers in the future.

8.2 Shortcomings and Future Work

There is still room for improvements to both the NEST mechanized formalism and the Elixir-based implementation of Sparrow. We now present future research opportunities.

Fully-fledged Sparrow formalisation. The current formalisation of Sparrow (NEST) only supports the core coordination abstractions of Sparrow. One avenue of future work would be to incorporate all of Sparrow's abstractions into NEST. This could lead us to find new requirements and constraints for our randomized message generation algorithm.

Generic message patterns test suites. In section 5.2.2, we have introduced randomized tests for NEST's patterns. This implementation targets the particular requirements of these patterns. However, to the best of our knowledge, there is no such kind of generic test suite for actor models. It would be interesting to explore the possibility to extend our *message generation algorithm* and *test abstractions* (i.e., `pattern-test` and `pattern-traces`) to support general actor-based language implementations.

Explicit unification of logic variables. Sparrow's patterns inherited the unification behaviour of Elixir's pattern-matching mechanism for its logic variables. This behaviour is desirable in many cases of composite patterns in order to reduce the need for more explicit filter expressions (e.g., guards). However, it can lead to unexpected result particularly by patterns that reuse definitions of other patterns. One avenue of future work could be to study a language design which allows only the unification of logic variables between anonymous patterns and requires an explicit unification expression in other cases.

Dynamic pattern engine. The current implementation of JuPITer requires knowing all patterns that will be part of its matching node's network at compile time. This was a decision that we made during its implementation, but there are no constraints imposed by RETE or Elixir that prevent adding or removing a pattern dynamically. It would be interesting for more dynamic cyber-physical systems to explore the possibility to add and remove patterns at runtime. At the same time, we need to evaluate the impact in terms of performance and security of such dynamic behaviour.

Performance. In the development of our DSL, we did not focus on efficiency. Nevertheless, we adopted a RETE-based matching mechanism for Sparrow's patterns to increase their matching performance based on historical evidence about RETE's benefits. A future task of our work may target the evaluation of Sparrow's composite patterns matching overhead concerning the matching process of traditional actor-based languages (e.g., Elixir). We could measure the *memory consumption* and the *matching speed* of the manual coordination patterns implemented in Elixir and the ones implemented in Sparrow. The result of this analysis could help us to determine the overhead introduced by the pattern engine of Sparrow's actors and whether that trade off is worth the additional expressiveness Sparrow provides.

Applicability to other concurrency models. The current implementation of Sparrow and NEST targets actors as their concurrency model. It would be interesting to explore the integration of the synchronization abstractions identified in this dissertation to other concurrency models (e.g., threads).

8.3 Closing Remarks

The fast-growing development of cyber-physical systems (CPS) in the last decade has introduced new technological challenges for the hardware and software industry. In this dissertation, we stated that the actor model constitutes a promising avenue to tackle software requirements of emerging challenges for CPS. Traditional actor-based languages have always embraced asynchronous point-to-point communication based on the exchange

of and reaction to single messages. However, the distributed and heterogeneous architecture of modern CPS, imposes complex synchronization patterns between the different components of a system. In this dissertation, we have shown that the message matching mechanism of traditional actor-based languages does not satisfy the current synchronization requirements of CPS. Although the actor model has been the subject of multiple extensions meant to improve its synchronization abstractions, there is still a serious need for more research to tackle the increased complexity of CPS. To the best of our knowledge Sparrow is the first attempt to tackle this issue by reconciling CEP techniques and Join Patterns with the message matching mechanism of the actor model.

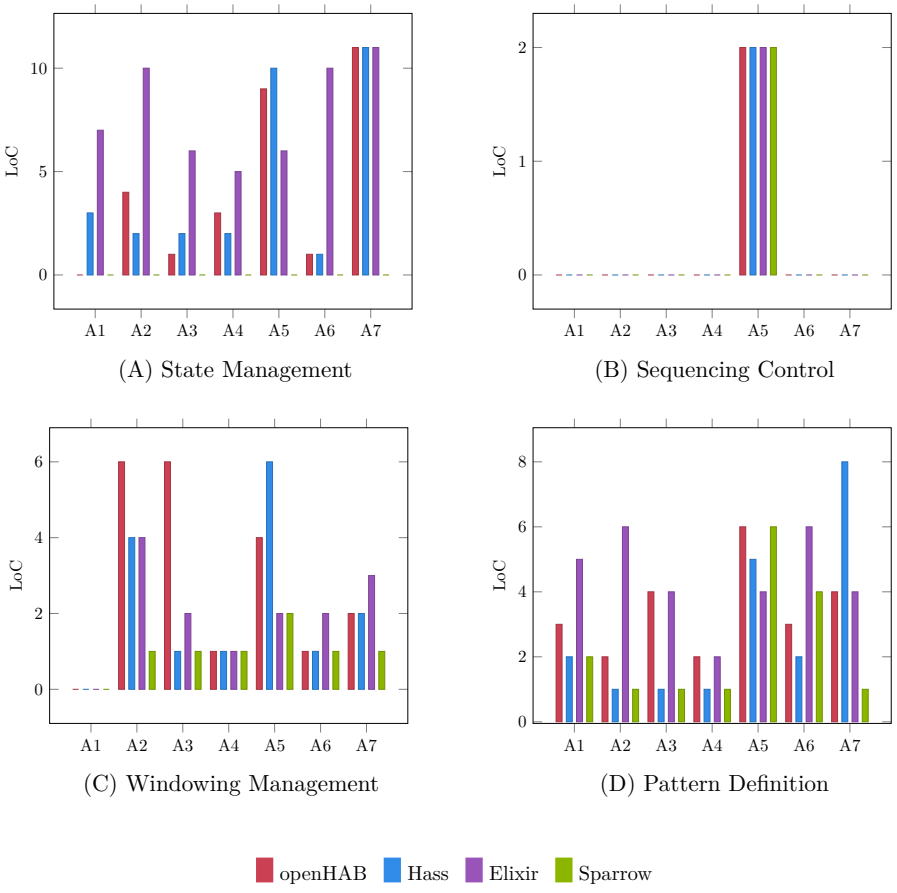
APPENDIX A

Appendices

A.1 LoC Breakdown of the Smart-Home Scenario Solutions

Table A.1 Overview of lines of code for the different scenarios according to the four identified coding concerns.

	openHAB	Hass	Elixir	Sparrow
State management				
Automation 1	0	3	7	0
Automation 2	4	2	10	0
Automation 3	1	2	6	0
Automation 4	3	2	5	0
Automation 5	9	10	6	0
Automation 6	1	1	10	0
Automation 7	11	11	11	0
Sequencing control				
Automation 1	0	0	0	0
Automation 2	0	0	0	0
Automation 3	0	0	0	0
Automation 4	0	0	0	0
Automation 5	2	2	2	2
Automation 6	0	0	0	0
Automation 7	0	0	0	0
Windowing management				
Automation 1	0	0	0	0
Automation 2	6	4	4	1
Automation 3	6	1	2	1
Automation 4	1	1	1	1
Automation 5	4	6	2	2
Automation 6	1	1	2	1
Automation 7	2	2	3	1
Pattern definition				
Automation 1	3	2	5	2
Automation 2	2	1	6	1
Automation 3	4	1	4	1
Automation 4	2	1	2	1
Automation 5	6	5	4	6
Automation 6	3	2	6	4
Automation 7	4	8	4	1
Total lines of code (LoC)	75	68	102	25



(A) State Management

(B) Sequencing Control

(C) Windowing Management

(D) Pattern Definition

■ **Figure A.1** Summary of analyzing different solutions for the seven scenarios

A.2 Source code of the Sparrow.Actor module

Listing A.1 Source code of the `Sparrow.Actor` module (Part 1)

```
1  defmodule Sparrow.Actor do
2    @moduledoc false
3    alias Sparrow.Core.{Builder, AppMonitor}
4    alias Sparrow.Engine.Jupiter
5    alias Sparrow.Exceptions.ReactionError
6
7    defmacro ___using___(_) do
8      storeName =
9        :crypto.strong_rand_bytes(20)
10       |> :base64.encode()
11       |> String.to_atom()
12
13     Builder.init(storeName)
14     quote do
15       import Sparrow.Actor
16       use GenServer
17
18       def start(options \\ [], linked \\ true) do
19         case linked do
20           true -> GenServer
21                   .start_link(___MODULE___, [], options)
22           false -> GenServer
23                   .start(___MODULE___, [], options)
24         end
25       end
26
27       ## Actor Server Callbacks
28       @impl true
29       def init(args) do
30         {:ok, args, {:continue, :init}}
31       end
32
33       @impl true
34       def handle_continue(:init,_args) do
35         app = Mix.Project.config()
36             |> Keyword.get(:app)
37         nodeId = self()
```

Listing A.2 Source code of the `Sparrow.Actor` module (Part 2)

```
38          state = Sparrow.Actor
39                  .___init_network(___MODULE___, app, nodeId)

40          {:noreply, state}
41        end
42
43        def stop(pid) do
44          GenServer.stop(pid)
45        end
46
47        def send(pid, message) do
48          GenServer.cast(pid, {:send, message})
49        end
50
51        @impl true
52        def handle_cast({:send, msg}, {engine, _} = state) do

53          Sparrow.Engine.Jupiter.process_message(engine, msg)

54          {:noreply, state}
55        end
56      end
57    end
58
59    defmacro pattern(expr)
60    defmacro pattern({name, _, [{:as, _, [pttr | []]}]}) do
61      Builder.build_pattern(name, pttr)
62      |> Macro.expand(___CALLER___)
63    end
64
65    defmacro pattern({name, _,
66                      [{:as, _, [pttr, _options | []] = p}]}) do

67      Builder.build_pattern(name, p)
68      |> Macro.expand(___CALLER___)
69    end
70
71    defmacro reaction({name, _, [header]}, do: body) do
72      Builder.build_reaction(name, header, body)
73      |> Macro.expand(___CALLER___)
```

Listing A.3 Source code of the `Sparrow.Actor` module (Part 3)

```
74    end
75    defmacro reaction(_call, _expr) do
76      raise ReactionError,
77        message:
78          "Invalid reaction definition.
79          A reaction only receives a single
80          argument of type `Sparrow.Engine.Token`."
81    end
82
83    defmacro react_to({pName, _, _}, with: {rName, _, _})
84            when is_atom(pName) and is_atom(rName) do
85      Builder.register_react_to(pName, rName)
86      |> Macro.expand(___CALLER___)
87    end
88
89    defmacro react_to(_, _) do
90      raise ReactionError,
91        message:
92          "Invalid reaction registration.
93          Check that both reaction and
94          pattern names are correct."
95    end
96
97    def ___init_network(module, app, nodeId) do
98      monitor = Application.get_env(app, :monitor, false)
99      ttl = Application.get_env(app, :ttl, {2, :hours})
100     AppMonitor.notify(:add_actor, nodeId, {app, module}, monitor)
101
102     {patterns, reactions, prRegistry} = ___dnetwork(module)
103
104     {:ok, engine} = Jupiter.start(patterns, reactions,
105                                   prRegistry, nodeId, ttl, monitor)
106     {engine, monitor}
107   end
```

Listing A.4 Source code of the `Sparrow.Actor` module (Part 4)

```
108   def ___dnetwork(module) do
109     functions = module.___info___(:functions)
110     patterns =
111       functions
112       |> Enum.filter(fn {f, _} ->
113         Atom.to_string(f)
114           |> String.starts_with?("___pattern_")
115       end)
116       |> Enum.map(fn {f, _} -> apply(module, f, []) end)
117       |> Enum.sort(&(&1.id < &2.id))
118
119     reactions =
120       functions
121       |> Enum.filter(fn {f, _} ->
122         Atom.to_string(f)
123           |> String.starts_with?("___reaction_")
124       end)
125       |> Enum.map(fn {f, _} -> apply(module, f, []) end)
126       |> Enum.reduce(%{},
127                   fn {n, f}, acc -> Map.put(acc, n, f) end)
128     prRegistry =
129       functions
130       |> Enum.filter(fn {f, _} ->
131         Atom.to_string(f)
132           |> String.starts_with?("___react_to_")
133       end)
134       |> Enum.map(fn {f, _} -> apply(module, f, []) end)
135       |> Enum.sort(&(elem(&1, 0) < elem(&2, 0)))
136       |> Enum.reduce(%{},
137                   fn {_, p, r}, acc ->
138                    group_reactions(acc, p, r)
139                   end)
140     {patterns, reactions, prRegistry}
141   end
142 end
```

A.3 Statistical Analysis for Pattern Definition

Listing A.5 Python script for pattern definition analysis

```
import pandas as pd

results = pd.read_csv('pattern_definition.csv')

openHAB = results[results.Platform.isin(['openHAB'])]
hass = results[results.Platform.isin(['Hass'])]
elixir = results[results.Platform.isin(['Elixir'])]
sparrow = results[results.Platform.isin(['Sparrow'])]

openHAB.LoC.describe()
#mean 3.428571, std 1.397276
#min 2.000000, 25% 2.500000
#50% 3.000000, 75% 4.000000
#max 6.000000

hass.LoC.describe()
#mean 2.857143, std 2.672612
#min 1.000000, 25% 1.000000
#50% 2.000000, 75% 3.500000
#max 8.000000

elixir.LoC.describe()
#mean 4.428571, std 1.397276
#min 2.000000, 25% 4.000000
#50% 4.000000, 75% 5.500000
#max 6.000000

sparrow.LoC.describe()
#mean 2.285714, std 1.976047
#min 1.000000, 25% 1.000000
#50% 1.000000, 75% 3.000000
#max 6.000000
```

A.4 Statistical Analysis for State Management

Listing A.6 Python script for state management analysis

```
1  import pandas as pd
2
3  results = pd.read_csv('state_management.csv')
4  openHAB = results[results.Platform.isin(['openHAB'])]
5  hass = results[results.Platform.isin(['Hass'])]
6  elixir = results[results.Platform.isin(['Elixir'])]
7  sparrow = results[results.Platform.isin(['Sparrow'])]
8
9  openHAB.LoC.describe()
10 #mean 4.142857, std 4.259443
11 #min 0.000000, 25% 1.000000
12 #50% 3.000000, 75% 6.500000
13 #max 11.000000
14
15 hass.LoC.describe()
16 #mean 4.428571, std 4.197505
17 #min 1.000000, 25% 2.000000
18 #50% 2.000000, 75% 6.500000
19 #max 11.000000
20
21 elixir.LoC.describe()
22 #mean 7.857143, std 2.410295
23 #min 5.000000, 25% 6.000000
24 #50% 7.000000, 75% 10.000000
25 #max 11.000000
26
27 sparrow.LoC.describe()
28 #mean 0.0, std 0.0
29 #min 0.0, 25% 0.0
30 #50% 0.0, 75% 0.0
31 #max 0.0
```

A.5 Statistical Analysis for Windowing Management

Listing A.7 Python script for windowing management analysis

```
1  import pandas as pd
2
3  results = pd.read_csv('windowing_management.csv')
4
5  openHAB = results[results.Platform.isin(['openHAB'])]
6  hass = results[results.Platform.isin(['Hass'])]
7  elixir = results[results.Platform.isin(['Elixir'])]
8  sparrow = results[results.Platform.isin(['Sparrow'])]
9
10 openHAB.LoC.describe()
11 #mean 2.857143, std 2.478479
12 #min 0.000000, 25% 1.000000
13 #50% 2.000000, 75% 5.000000
14 #max 6.000000
15
16 hass.LoC.describe()
17 #mean 2.142857, std 2.115701
18 #min 0.000000, 25% 1.000000
19 #50% 1.000000, 75% 3.000000
20 #max 6.000000
21
22 elixir.LoC.describe()
23 #mean 2.000000, std 1.290994
24 #min 0.000000, 25% 1.500000
25 #50% 2.000000, 75% 2.500000
26 #max 4.000000
27
28 sparrow.LoC.describe()
29 #mean 1.00000, std 0.57735
30 #min 0.00000, 25% 1.00000
31 #50% 1.00000, 75% 1.00000
32 #max 2.00000
```

A.6 Normalization of LoC

Listing A.8 LoC Normalization Script in Elixir

```
upper_range = 5

normalize = fn (list, upper_range) ->
  min = Enum.min list
  max = Enum.max list
  Enum.reduce(list, [], fn x, acc -> acc++[
  Float.round(((x-min)/(max-min))*upper_range, 2)] end)
end

ajust = fn (list) ->
  Enum.map(list, fn x ->
    cond do
      x == 0 -> Float.round(x+0.01, 2)
      x < (upper_range-1) -> Float.round(x+1, 2)
      true -> x
    end
  end)
end

all_values = [
29,2,20,24,
31,2,15,20,
55,2,12,31,
0,2,7,16]

all_values
|> normalize.(upper_range)
|> ajust.()
|> IO.inspect()
```

A.7 PLT Redex in a Nutshell

PLT Redex (or Redex for short) [21] is a domain-specific language in Racket for formalizing operational semantics with powerful pattern-matching capabilities. It allows language designers to write down the language grammar, reduction rules, and relevant meta-functions for their calculi in a single language. Besides its language capabilities, Redex offers language designers a set of tools integrated into the DrRacket IDE, including a stepper for small-step operational semantics, hand-written/randomized test suites, and inspectors reduction graphs. In short, Redex aims to help language designers to validate their language implementations against their specifications at a low cost.

A.7.0.1 Language Definition

Listing A.9 shows a simple example of a language definition and its grammars in Redex. The language `Bool` has two non-terminal grammars `exp` and `val`. The first non-terminal has associated three productions or patterns representing all valid expressions valid in the `Bool` language. The second one has two patterns that match boolean values. Notice that Redex uses a parenthesized version of BNF notation to define its tree grammars. Furthermore, it does not use vertical bars to separate productions, simply juxtaposing them instead.

Listing A.9 Example of a language definition

```
1 (define-language Bool
2   (exp ::= val
3            (exp && exp)
4            (exp || exp))
5   (val ::= #true
6            #false))
```

Language designers can use the built-in function `(redex-match? L p t)` to test their languages' syntax. This function determines if the term `t` matches the pattern `p` in the language `L`. Listing A.10 shows how to check the syntax of both Bool's non-terminals using the above function. The underscore (_) symbol is used to refer to an instance of a non-terminal, also known as *named non-terminals*. Furthermore, the pattern `p` can refer to particular productions (see line 3) or the global non-terminal element (see lines 1, 2, 4).

Listing A.10 Example of syntax checks

```
1 (redex-match? Bool val_a (term #true))
  ;; #true
2 (redex-match? Bool exp_a (term (#true || #false)))
  ;; #true
3 (redex-match? Bool (exp_a && exp_b) (term (#true || #true)))

  ;; #false
4 (redex-match? Bool exp_a (term 1))
  ;; #false
```

A.7.0.2 Hand-written and Randomized Unit Tests

Using Redex's unit testing library, we can transform the syntax check examples of Listing A.10 in proper tests. Listing A.11 shows an example of a test unit module with four *hand-written* tests (see lines 4-7) and their results (text in green color[1]). This example uses an auxiliary function `valid-syntax?` (see line 2) to compact the invocations of the built-in `test-equal` function. The tests' results are shown in the terminal by invoking the built-in `test-results` function (see line 9). As we can observe, the last test (see line 7) fails because any of the defined productions of the non-terminal `exp` matches the term `1`.

Listing A.11 Example of hand-written unit tests

```
1 (module+ test
2   (define valid-syntax? (redex-match? Bool exp_i))
3
4   (test-equal (valid-syntax? (term #true)) #true)
5   (test-equal (valid-syntax? (term (#true && #true))) #true)

6   (test-equal (valid-syntax? (term (#true || #false))) #true)

7   (test-equal (valid-syntax? (term 1)) #true)
8
9   (test-results))
;; ####### OUTPUT ######
;; FAILED /../nest-plt-redex/bool_lang.rkt:33.2
```

[1]The font color of comments and terminal outputs will always be green. The text will always start with a double comment symbol of its language (e.g., ;;)

```
;; actual: #f
;; expected: #t
;; 1 test failed (out of 4 total).
```

Hand-written tests are useful for initial checks of the language syntax. However, they require extra work from designers, and they cannot exhaustively test a language. Redex supports *randomized* tests of syntactic properties using the `redex-check` function to overcome this issue. This built-in function searches for a counterexample to the grammar productions. It uses it a "guess and check" strategy to freely generates candidate terms and tests whether they happen to satisfy the production constraints. Listing A.12 shows a randomized test for the non-terminal `exp`, and its output. Our calculus makes use of both types of unit tests to check its syntax exhaustively.

Listing A.12 Example of a randomized test

```
1 (redex-check
2    Bool
3    exp_i
4    (redex-match? Bool exp (term exp_i))
5    #:attempts 1000)
 ;; redex-check: /../nest-plt-redex/bool_lang.rkt:40
 ;; no counterexamples in 1000 attempts
```

A.7.0.3 Judgment Forms and Metafunctions

Redex offers an easy way to define a relation on terms through a *judgment form*. Judgments can check various types of relations, including type inference rules and well-formedness conditions. However, they cannot express inference rules that require guessing [21]. Listing A.13 shows an example of a judgment `Equals?` which defines an equality relation on booleans values. Like other Redex's a !22primitives, the first argument of the `define-judgment-form` function refers to the *language*. The next two arguments specify a *mode* and a *contract* for the judgment. The former (see line 2) specifies the number of expected input terms (i.e., two). The latter (see line 3) specifies that both input terms must match the non-terminal `val`. Finally, this judgment receives a set of inferences rules (see lines 4-9), named `BothTrue?` and `BothFalse?`. In this simple example, both rules hold if their arguments have the same value. Additionally, rules

Listing A.13 Example of a judgment form definition

```
1  (define-judgment-form Bool
2   #:mode (Equals? I I)
3   #:contract (Equals? val val)
4   [
5    --- BothTrue?
6    (Equals? #true #true)]
7   [
8    --- BothFalse?
9    (Equals? #false #false)])
10
11  (judgment-holds (Equals? #true #true))
    ;; #true
13  (judgment-holds (Equals? #false #false))
    ;; #true
14  (judgment-holds (Equals? #true #false))
    ;; #false
```

can use `where` clauses (a.k.a guards) to check that each pattern matches the corresponding term before applying it. Language designers can test the above inference rules by using the built-in function `judgment-holds`. Lines 11-13 showcase the use of this primitive to check if the judgments "`#true #true`", "`#false #false`", and "`#true #false`" satisfy any of the patterns defined by the judgment form `Equals?`.

In addition to judgment forms, Redex supports term-level functions, called *metafunctions*. Metafunctions like judgments are a formal alternative to escape from Redex (a.k.a Redex mode) to Racket (a.k.a Racket mode). Listing A.14 shows the equivalent metafunction to the judgment form defined in Listing A.13. The significant difference between the listings mentioned above is the addition of a new rule (see Listing A.14 lines 12-14). Without this rule, the Redex's interpreter will raise an error since no clause will match the terms "#true #fase" (see line 18). Although metafunctions and judgment forms can be used interchangeable, the latter are preferred since they are easier to read, debug, and maintain.

Listing A.14 Example of a metafunction definition

```
1  (define-metafunction Bool
2    Equals? : val val -> boolean
3    [
4     ;--- BothTrue
5     (Equals? #true #true)
6     #true]
7    [
8     ;--- BothFalse
9     (Equals? #false #false)
10    #true]
11   [
12    ;--- Otherwise
13    (Equals? val_0 val_1)
14    #false])
15
16  (term (Equals? #true #true))
    ;; #true
17  (term (Equals? #false #false))
    ;; #true
18  (term (Equals? #true #false))
    ;; #false
```

A.7.0.4 Reduction Relations

In Redex, a reduction relation is a set of term-rewriting rules. Language designers can choose to apply one step of a reduction relation or reduce it until its normal form or some condition. A language must include at least three non-terminals to apply a reduction relation. First, a non-terminal for the reduction relation domain (see listing A.9 line 2-4). Second, a non-terminal for a subset of the domain cannot be reduced further (see listing A.9 line 5-6). Third, a non-terminal for evaluation contexts. An evaluation context is a special term that contains a hole. Reduction relations can match a term against an evaluation context to find a subterm that matches the hole. Our definition of the `Bool` language (see Listing A.10) covers the first two required non-terminals, but it does not contain any evaluation contexts.

Listing A.15 illustrates how to extend the `Bool` language to add its evaluation contexts using the primitive `define-extended-language`. The

Listing A.15 Add evaluation contexts to an existent language

```
1 (define-extended-language Bool+ Bool
2   (E ::= hole
3          (E && exp)
4          (E || exp)))
```

`Bool+` language adds a new non-terminal `E` and inherits all non-terminals of its parent `Bool`. The non-terminal `E` contains three patterns that represent the evaluation contexts of the `Bool+` language. The first pattern will try to match any subexpression using the hole abstraction, while the other two patterns will try to match conjunctions and disjunctions expressions.

Listing A.16 shows the reduction relation for both conjunction (see lines 4-8) and disjunction (see lines 9-13) expressions supported by the `Bool+` language. Like other abstractions, the `reduction-relation` primitive needs a language to know how to interpret patterns, a domain to specify the pattern that its input/output must match and a set of rewrite rules. The domain in the above reduction rule is the non-terminal `exp`, in which a term represents a simple (e.g., #true) or composed (e.g., a conjunction) boolean expression. Simple boolean expressions are values (see the non-terminal *val*) that cannot reduce further. The use of `in-hole` primitive in both cases (i.e., "and-step" and "or-step") of the reduction relation `bool-step` is two fold. In one hand, as the first argument of `->`, it searches a term for a subterm that Redex can apply a reduction rule to (see lines 4, 9). On the other hand, as the second argument of `->`, it puts a new value back into the hole in the evaluation context (see lines 5, 10).

Redex also gives language designers access to a non-deterministic interpreter to evaluate their reduction relations, build reduction graphs (via traces), and detects cycles in them. For example, at the end of Listing A.16 (see lines 15-18) the function `apply-reduction-relation` applies the reduction relation `bool-step` to a term and returns a list of ways that the term can step. The cases of reduction relation form a set, not a sequence. Consequently, if a term matches more than one case, Redex will apply all of them.

Listing A.16 Example of a reduction relation definition and evaluation

```
(define bool-step
  (reduction-relation Bool+
    #:domain exp
    [--> (in-hole E (val_lhs && val_rhs))
      (in-hole E val_new)
      "and-step"
      (where val_new
            ,(and (term val_lhs) (term val_rhs)))]

    [--> (in-hole E (val_lhs || val_rhs))
      (in-hole E val_new)
      "or-step"
      (where val_new
            ,(or (term val_lhs) (term val_rhs)))]))


(apply-reduction-relation
    bool-step (term #true))
;; '()
(apply-reduction-relation
    bool-step (term (#true && #true)))
;; '(#true)
(apply-reduction-relation
    bool-step (term (#true && #false)))

;; '(#false)
(apply-reduction-relation
    bool-step (term ((#true || #false) || #true)))

;; '((#true || #true))
```

A.8 NEST Semantics in Redex

Listing A.17 Source code of the NEST (Part 1)

```
1  #lang racket
2
3  (provide NEST NEST-R NEST-T) ;; Language

4
5  (require redex)
6
7  ;; NEST's BNF grammar
8  (define-language NEST
9    (pe ::= (pattern pn p))
10   (re ::= (reaction rn e))
11   (ae ::= (actor an (react-to pn rn) ...))

12   (e ::= nil
13          x l i
14          (lambda [e ...] e)
15          (let (x e) in e)
16          (spawn an)
17          (send e e)
18          (react-to pn rn)
19          (remove rn pn)
20          (remove-reactions pn)
21          (e ...)
22          (~ e ...)
23          (aop e e)
24          (lop e e)
25          (cop e e)
26          v)
27
28   (aop ::= + - / *)
29   (lop ::= and andThen or)
30   (cop ::= == >= <= > <)
31   (v ::= nil number integer string boolean atom)

32   (atom ::= (variable-prefix :))
33   (g ::= (when e))
34
35   (q ::= (m ...))
```

Listing A.18 Source code of the NEST (Part 2)

```
36    (m ::= (x (t v ...)))
37    (p ::= pb
38           (pb g))
39
40    (pb ::= ep
41            (lop pb pb))
42
43    (ep ::= s
44            (s po pt ...)
45            (not s (window number u)))
46
47    (t ::= atom)
48
49    (s ::= (t att ...))
50
51    (att ::= x v)
52
53    (po ::= (count integer)
54            (every integer)
55            (window number u)
56            (debounce integer u)
57    )
58
59    (pt ::= (fold e) (map e) (bind x))
60
61    ;; time units
62    (u ::= secs mins hours days weeks)
63
64    (pl ::= ((pn . (p (m ...))) ...))
65
66    (rl ::= ((rn . (e)) ...))
67    (pr ::= ((pn . (rn ...)) ...))
68    (ml ::= ((x . (x)) ...))
69
70    (x l i pn rn an ::= variable-not-otherwise-mentioned)

71    (id ::= variable-not-otherwise-mentioned)

72  )
```

Listing A.19 Source code of the NEST (Part 3)

```
(define-extended-language NEST-R NEST
 (e ::= .... pe re)
 (g ::= .... nil)
 (v ::= .... rf)

 (rf ::= (ref id))

 (K ::= (a ... A a ...))
  (A ::=
     hole
     (pl rl (actor id q pr E)))

  (E ::=
     hole
     (v ... E e ...)
     (let (x E) in e)
     (send E e)
     (send v E)
     (aop E e)
     (aop v E)
     (cop E e)
     (cop v E)
)


  (k ::= (a ...))
  (a ::= (pl rl (actor id q pr e))
         ((pl rl (actor id q pr e))))

 )

(define-extended-language NEST-T NEST-R

  (m ::= (v (t v ...)))
  )
```

Bibliography

[1] AB, E. Otp design principles. URL: https://erlang.org/doc/design_principles/des_princ.html. Accessed: 2019-05-21.

[2] AB, E. Dialyzer documentation. URL: https://erlang.org/doc/man/dialyzer.html, 2021. Accessed: 2021-02-08.

[3] Abd Alrahman, Y., De Nicola, R., and Loreti, M. On the power of attribute-based communication. In *Formal Techniques for Distributed Objects, Components, and Systems* (2016), vol. 9688 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg.

[4] acatech (Ed.). *Cyber-Physical Systems. Driving force for innovation in mobility, health, energy and production (acatech POSITION PAPER)*. Heidelberg et al.: Springer Verlag, 2011.

[5] Adi, A., and Etzion, O. Amit - the situation manager. *The VLDB Journal 13*, 2 (2004), 177–203.

[6] AG, C. Networked mobility. URL: https://www.continental.com/en/products-and-innovation/innovation/connectivity/networked-mobility, 2021. Accessed: 2021-04-28.

[7] Agha, G., and Callsen, C. J. Actorspace: An open distributed programming paradigm. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 1993), PPOPP '93, ACM, pp. 23–32.

[8] Akdere, M., undefinedetintemel, U., and Tatbul, N. Plan-Based Complex Event Detection across Distributed Sources. *Proc. VLDB Endow. 1*, 1 (2008), 66–77.

[9] Anicic, D., Fodor, P., Rudolph, S., Stühmer, R., Stojanovic, N., and Studer, R. *ETALIS: Rule-Based Reasoning in Event Processing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 99–124.

[10] Arbab, F. Reo: a channel-based coordination model for component composition. *Math. Struct. Comput. Sci. 14*, 3 (2004), 329–366.

[11] Barga, R. S., and Caituiro-Monge, H. Event correlation and pattern detection in cedr. In *Current Trends in Database Technology – EDBT 2006* (Berlin, Heidelberg, 2006), vol. 4254 of *Coordination Languages and Models*, Springer Berlin Heidelberg, pp. 919 – 930.

[12] Benton, N., Cardelli, L., and Fournet, C. Modern concurrency abstractions for c#. In *ECOOP 2002 - Object-Oriented Programming* (2002), B. Magnusson, Ed., vol. 2374 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 415–440.

[13] Clebsch, S., Drossopoulou, S., Blessing, S., and McNeil, A. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (New York, NY, USA, 2015), AGERE! 2015, Association for Computing Machinery, p. 1–12.

[14] Cugola, G., and Margara, A. Raced: An adaptive middleware for complex event detection. In *Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware* (New York, NY, USA, 2009), ARM '09, Association for Computing Machinery.

[15] Cugola, G., and Margara, A. Tesla: A formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems* (New York, NY, USA, 2010), DEBS '10, Association for Computing Machinery, p. 50–61.

[16] Cugola, G., and Margara, A. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv. 44*, 3 (June 2012).

[17] De Nicola, R., Duong, T., Inverso, O., and Trubiani, C. Aerlang: Empowering erlang with attribute-based communication. *Science of Computer Programming 168* (2018), 71 – 93.

[18] Demers, A., Gehrke, J., Hong, M., Riedewald, M., and White, W. Towards expressive publish/subscribe systems. In *Proceedings of the 10th International Conference on Advances in Database Technology* (Berlin, Heidelberg, 2006), EDBT'06, Springer-Verlag, p. 627–644.

[19] Dinges, P., and Agha, G. Scoped synchronization constraints for large scale actor systems. In *Coordination Models and Languages* (Berlin, Heidelberg, 2012), M. Sirjani, Ed., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 89–103.

[20] Eugster, P., and Jayaram, K. R. Eventjava: An extension of java for event correlation. In *ECOOP 2009 – Object-Oriented Programming* (Berlin, Heidelberg, 2009), Springer Berlin Heidelberg, pp. 570–594.

[21] Felleisen, M., Findler, R. B., and Flatt, M. *Semantics Engineering with PLT Redex*, 1 ed. The MIT Press, 2009.

[22] Forgy, C. L. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence 19*, 1 (1982), 17–37.

[23] Fowler, S. An erlang implementation of multiparty session actors. In *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016* (2016), M. Bartoletti, L. Henrio, S. Knight, and H. T. Vieira, Eds., vol. 223 of *EPTCS*, pp. 36–50.

[24] Frølund, S., and Agha, G. A language framework for multi-object coordination. In *ECOOP' 93 — Object-Oriented Programming* (Berlin, Heidelberg, 1993), O. M. Nierstrasz, Ed., Springer Berlin Heidelberg, pp. 346–360.

[25] Frølund, S., and Agha, G. Abstracting interactions based on message sets. In *Object-Based Models and Languages for Concurrent Systems* (Berlin, Heidelberg, 1995), P. Ciancarini, O. Nierstrasz, and A. Yonezawa, Eds., Springer Berlin Heidelberg, pp. 107–124.

[26] Ganssle, J. G. A guide to debouncing. url: `http://www.ganssle.com/debouncing.htm`. Accessed: 2020-10-01.

[27] Garnock-Jones, T., and Felleisen, M. Coordinated concurrent programming in syndicate. In *Programming Languages and Systems*

(Berlin, Heidelberg, 2016), P. Thiemann, Ed., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 310–336.

[28] Gelernter, D. Generative communication in linda. *ACM Trans. Program. Lang. Syst. 7*, 1 (Jan. 1985), 80–112.

[29] Geng, H., and Jamali, N. *interActors: A Model for Separating Communication Concerns of Concurrent Systems.* Lecture Notes in Computer Science. Springer International Publishing, Cham, 2018, pp. 186–215.

[30] Gutierrez-Nolasco, S., and Venkatasubramanian, N. A reflective middleware framework for communication in dynamic environments. In *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE* (Berlin, Heidelberg, 2002), R. Meersman and Z. Tari, Eds., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 791–808.

[31] Gyllstrom, D., Agrawal, J., Diao, Y., and Immerman, N. On supporting kleene closure over event streams. In *2008 IEEE 24th International Conference on Data Engineering* (April 2008), pp. 1391–1393.

[32] Haller, P., and Van Cutsem, T. Implementing joins using extensible pattern matching. In *Coordination Models and Languages.* Springer Berlin Heidelberg, Berlin, Heidelberg, June 2008, pp. 135–152.

[33] Harrison, J. Runtime type safety for erlang/otp behaviours. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang* (New York, NY, USA, 2019), Erlang 2019, Association for Computing Machinery, p. 36–47.

[34] Hewit, C., and Smith, B. A plasma primer (draft). *AI Lab Working Paper 92, MIT* (1975).

[35] Hewitt, C., Bishop, P., and Steiger, R. A universal modular actor formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (San Francisco, CA, USA, 1973), IJCAI'73, Morgan Kaufmann Publishers Inc., p. 235–245.

[36] Honda, K., Yoshida, N., and Carbone, M. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2008), POPL '08, Association for Computing Machinery, p. 273–284.

[37] IMEC. City of things. Zenodo. Jul. 2019. doi: 10.5281/zenodo.3972329.

[38] Inc., E. Esper documentation. url: https://www.nasa.gov/nh/pluto-the-other-red-planet, 2015. Accessed: 2020-10-26.

[39] Kambona, K. *Reentrancy & Scoping in Rule Engines for Cloud-based Applications.* PhD thesis, Vrije Universiteit Brussel, Brussels, BE, 06 2018.

[40] Khosravi, R., and Sabouri, H. Using coordinated actors to model families of distributed systems. In *Coordination Models and Languages* (Berlin, Heidelberg, 2012), M. Sirjani, Ed., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 74–88.

[41] Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J. A., Rafkind, J., Tobin-Hochstadt, S., and Findler, R. B. Run your research: On the effectiveness of lightweight mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2012), POPL '12, Association for Computing Machinery, p. 285–296.

[42] Laichi, B., and Sami, Y. Atc: actors with temporal constraints. In *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC 2001* (2001), IEEE Computer Society, pp. 306–313.

[43] Li, G., and Jacobsen, H.-A. Composite subscriptions in content-based publish/subscribe systems. In *Middleware 2005* (Berlin, Heidelberg, 2005), G. Alonso, Ed., Springer Berlin Heidelberg, pp. 249–269.

[44] Mei, Y., and Madden, S. Zstream: A cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*

(New York, NY, USA, 2009), SIGMOD '09, Association for Computing Machinery, p. 193–206.

[45] Minsky, N. H., and Ungureanu, V. Regulated coordination in open distributed systems. In *Coordination Languages and Models* (Berlin, Heidelberg, 1997), D. Garlan and D. Le Métayer, Eds., vol. 1282 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 81–97.

[46] Neykova, R., and Yoshida, N. Multiparty session actors. In *Coordination Models and Languages* (Berlin, Heidelberg, 2014), E. Kühn and R. Pugliese, Eds., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 131–146.

[47] Paula, I. S. Jcthorn - extending thorn with joins and chords. Master's thesis, Imperial College London, Department of Computing Imperial College London, June 2010.

[48] Pietzuch, P. R., Shand, B., and Bacon, J. Composite event detection as a generic middleware extension. *IEEE Network 18*, 1 (2004), 44–55.

[49] Plociniczak, H., and Eisenbach, S. Jerlang: Erlang with joins. In *Coordination Models and Languages: 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings* (Berlin, Heidelberg, 2010), D. Clarke and G. Agha, Eds., Springer Berlin Heidelberg, pp. 61–75.

[50] Prokopec, A., and Odersky, M. Isolates, channels, and event streams for composable distributed programming. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (New York, NY, USA, 2015), Onward! 2015, Association for Computing Machinery, p. 171–182.

[51] Qi, J., Kim, Y., Chen, C., Lu, X., and Wang, J. Demand response and smart buildings: A survey of control, communication, and cyber-physical security. *ACM Trans. Cyber-Phys. Syst. 1*, 4 (Oct. 2017).

[52] Ren, S., Agha, G. A., and Saito, M. A modular approach to programming distributed real-time systems. *Journal of Parallel and Distributed Computing 36*, 1 (1996), 4 – 12.

[53] Ren, S., Yu, Y., Chen, N., Marth, K., Poirot, P.-E., and Shen, L. Actors, roles and coordinators — a coordination model for open distributed and embedded systems. In *Coordination Models and Languages* (Berlin, Heidelberg, 2006), P. Ciancarini and H. Wiklicky, Eds., Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 247–265.

[54] Renaux, T. *A Distributed Logic Reactive Programming Model and its Application to Monitoring Security.* PhD thesis, Vrije Universiteit Brussel, Brussels, BE, 03 2019.

[55] Riedl, M., Zipper, H., Meier, M., and Diedrich, C. Automation meets cps. *IFAC Proceedings Volumes 46*, 7 (2013), 216–221. 11th IFAC Workshop on Intelligent Manufacturing Systems.

[56] Rodriguez Avila, H. Advanced Join Patterns for the Actor Model based on CEP Techniques (Scenarios solutions). Zenodo. Aug. 2019. doi: 10.5281/zenodo.3971130.

[57] Rodriguez Avila, H. Hass topics. Zenodo Jul. 2019. doi: 10.5281/zenodo.3611271.

[58] Rodriguez Avila, H. Home assistant - automation scenarios poll. Zenodo. Aug. 2019. doi: 10.5281/zenodo.3465385.

[59] Rodriguez Avila, H. Hubitat - automation scenarios poll. Zenodo. Aug. 2019. doi: 10.5281/zenodo.3464966.

[60] Rodriguez Avila, H. Openhab - automation scenarios poll. Zenodo. Aug. 2019. doi: 10.5281/zenodo.3666325.

[61] Rodriguez Avila, H. Openhab topics. Zenodo. Jul. 2019. doi: 10.5281/zenodo.3611168.

[62] Rodriguez Avila, H. Smartthings - automation scenarios poll. Zenodo. Aug. 2019. doi: 10.5281/zenodo.3464952.

[63] Rodriguez Avila, H. Statistic analysis of seven smart-home scenarios implemented in openHAB, Hass, Elixir, and Sparrow, Nov. 2020.

[64] Scalas, A., and Yoshida, N. Lightweight session programming in scala. In *30th European Conference on Object-Oriented Programming, ECOOP 2016* (2016), B. Lerner and S. Krishnamurthi, Eds., vol. 56 of *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 21:1–21:28.

[65] Scalas, A., Yoshida, N., and Benussi, E. Effpi: Verified message-passing programs in dotty. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala* (New York, NY, USA, 2019), Scala '19, Association for Computing Machinery, p. 27–31.

[66] Scholliers, C. *Ambient Contracts*. PhD thesis, Vrije Universiteit Brussel, Brussels, BE, 03 2013.

[67] Scholliers, C., Boix, E. G., and De Meuter, W. Totam: Scoped tuples for the ambient. *Electronic Communications of the EASST 19* (2009).

[68] Song, M., and Ren, S. Coordination operators and their composition under the actor-role-coordinator (arc) model. *SIGBED Review 8*, 1 (Mar. 2011), 14–21.

[69] Sulzmann, M., Lam, E. S. L., and Weert, P. Actors with multi-headed message receive patterns. In *Coordination Models and Languages*. Springer Berlin Heidelberg, 12 2008, pp. 315–330.

[70] Team, A. F. C. Flinkcep - complex event processing for flink. URL: `https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html`, 2020. Accessed: 2020-10-26.

[71] Team, C. Coq homepage. URL: `https://coq.inria.fr`, 2021. Accessed: 2021-02-11.

[72] Team, E. C. Guards - official documentation. Zenodo. Jul. 2019. DOI: `10.5281/zenodo.3971124`.

[73] Team, E. C. Erlang homepage. URL: `https://www.erlang.org`, 2020. Accessed: 2020-10-26.

[74] Team, S. Scribble homepage. URL: http://www.scribble.org, 2019. Accessed: 2019-08-19.

[75] Team, S. C. Scala homepage. URL: https://www.scala-lang.org, 2020. Accessed: 2020-10-26.

[76] Team, T. E. Elixir homepage. URL: https://elixir-lang.org, 2020. Accessed: 2020-10-26.

[77] Team, T. E. Supervisor behaviour. URL: https://hexdocs.pm/elixir/Supervisor.html, 2020. Accessed: 2020-10-26.

[78] Team, T. R. Racket homepage. URL: https://racket-lang.org, 2020. Accessed: 2020-09-22.

[79] Tomlinson, C., Kim, W., Scheevel, M., Singh, V., Will, B., and Agha, G. Rosette: An object-oriented concurrent systems architecture. *SIGPLAN Not. 24*, 4 (Sept. 1988), 91–93.

[80] Trono, J. A new exercise in concurrency. *ACM SIGCSE Bulletin 26* (09 1994), 8–10.

[81] Trottier-Hebert, F. Syntax in functions. pattern matching. URL: https://learnyousomeerlang.com/syntax-in-functions, 2013. Accessed: 2021-03-18.

[82] Van Cutsem, T., Gonzalez Boix, E., Scholliers, C., Lombide Carreton, A., Harnie, D., Pinte, K., and De Meuter, W. Ambienttalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures 40*, 3 (2014), 112 – 136.

[83] Van Ham, J. M., Salvaneschi, G., Mezini, M., and Noyé, J. Jescala: Modular coordination with declarative events and joins. In *Proceedings of the 13th International Conference on Modularity* (New York, NY, USA, 2014), MODULARITY '14, Association for Computing Machinery, p. 205–216.

[84] Varela, C., and Agha, G. A hierarchical model for coordination of concurrent activities. In *Coordination Languages and Models* (Berlin, Heidelberg, 1999), P. Ciancarini and A. L. Wolf, Eds., vol. 1594

of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 166–182.

[85] Varela, C., and Agha, G. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Notices 36*, 12 (Dec. 2001), 20–34.

[86] Wu, E., Diao, Y., and Rizvi, S. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2006), SIGMOD '06, Association for Computing Machinery, p. 407–418.

[87] Yonezawa, A., Briot, J.-P., and Shibayama, E. Object-oriented concurrent programming in abcl/1. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (New York, NY, USA, 1986), OOPSLA '86, Association for Computing Machinery, p. 258–268.

[88] Zhang, Y., Qiu, M., Tsai, C.-W., Hassan, M., and Alamri, A. Health-cps: Healthcare cyber-physical system assisted by cloud and big data. *IEEE Systems Journal 11* (08 2015), 1–8.

[89] Zimmer, D., and Unland, R. On the semantics of complex events in active database management systems. In *Proceedings of the 15th International Conference on Data Engineering* (Washington, DC, USA, 1999), ICDE '99, IEEE Computer Society, pp. 392–399.