ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

Master Degree in Computer Science and Engineering

Validation of FRASP specifications through controlled reactive simulations

Master thesis in:
Pervasive Computing

Supervisor
Prof. Mirko Viroli

Cosupervisor
Dott. Roberto Casadei
Dott. Gianluca Aguzzi

Candidate
Jahrim Gabriele Cesario

IV Graduation Session
Academic year 2022-2023

Abstract

Jahrim Gabriele Cesario: What is this thesis about? Max 2000 characters, strict.

Jahrim Gabriele Cesario: Optional. Max a few lines.

Acknowledgements

Jahrim Gabriele Cesario: Aknowledgements... Max 1 page.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

This chapter provides a summary of the content and organization of this thesis, describing the context, motivations and high-level goals of this project and how they are presented in this document.

1.1 Content

The ever-increasing availability of devices is creating an emerging class of distributed systems, called collective adaptive systems, with application domains such as smart cities, complex sensor networks and the Internet Of Things (IoT) [VAB+18]. The complexity of these systems calls for new programming paradigms better suited for large-scale distributed systems, such as aggregate computing [VBD+19].

Most state-of-the-art aggregate computing frameworks rely on a round-based computation model, which is simple, but limited in terms of flexibility and efficiency. To provide for the shortcomings of the round-based computation model, new reactive approaches are currently in research, such as the FRASP library [CDA+23], which is the subject of this work.

FRASP provides a novel domain-specific language for combining the functional reactive programming paradigm with the aggregate computing paradigm, extending the former to be applied in distributed systems and in particular in collective adaptive systems, while also extending the latter with a reactive computation model, replacing the typical round-based one.

At the time of writing, FRASP is a research project and there are many ideas, challenges and features still to be explored. However, the library requires a consolidated test suite before further evolution, to assess the correctness of its current implementation and prevent possible software regressions, that may be due to unsuspected interactions between present and future features.

The main goal of this thesis is to implement a validated version of the FRASP library, providing a clearer definition of its functionalities, verified through adequate unit tests. Properties concerning FRASP programs will be mainly evaluated via simulation, requiring a thorough analysis and validation of the current simulator as well.

1.2 Structure

The content of the thesis will be presented in detail in the following chapters.

First, Chapter 2 provides an overview of the main concepts and technologies used in this project, so that this document may be self-contained.

Jahrim Gabriele Cesario: TODO: describe other chapters...

1.3 Style

The writing style adopted within this document provides intentional meaning to the font styles used in words or sentences. Here follows a comprehensive list of such font styles and their meanings:

- *Italic*: used to draw the reader's attention towards certain words or sentences.
- **Bold**: used to introduce a new concept that has never been mentioned before in the document.
- `Monospace`: used to reference an existing concept in the source code of the project.

1.4 Prerequisites

The following chapters may contain references to concepts related to object-oriented and functional programming, assuming that the reader is familiar with such paradigms (specifically the Java [Ora] and Scala [Cen] documentations). Indeed, Scala has been adopted as the language of choice in this document for abstracting over software interfaces and writing pseudo-code, due to its clean and minimalistic functional syntax.

Chapter 2

Background

This chapter provides an overview of the main concepts and technologies used in this project, so that this document may be self-contained.

2.1 Concepts

This section provides a general technology-agnostic description of the main concepts referenced by this project.

2.1.1 Collective Adaptive Systems

Collective systems are distributed situated systems composed of a potentially large set of computing components, that are competing or cooperating to achieve a specific goal, interacting with each other and adapting to the changes of their environment [Fer15].

The behavior of a collective system as a whole is an expression of **collective intelligence** or **swarm intelligence**, in fact it emerges from the behaviors of its individual components, the local interactions between them and with their environment [LP00]. These concepts come from the study of self-organizing groups of entities in nature (e.g. ant colonies, bird flocks) applied to computer science, in pursuit of **adaptiveness** and in particular **self-organization**.

Adaptiveness is the ability of a system to change its behavior depending on the circumstances to better achieve its goals and it is so important in the applications of collective systems that they are often called directly **Collective Adaptive Systems (CASs)**. In fact, adaptiveness grants collective systems with the robustness needed to address unforeseen changes in operating conditions, which are typical in real-world environments (e.g. network failures, open networks, mobile components).

General adaptiveness can be obtained using different strategies, including centralized approaches in which a designated control system changes the behaviors of the system components depending on their perception of the local environment. However, CASs achieve adaptiveness specifically through a decentralized approach called self-organization, in which complex global ordered structures (e.g. collective behaviors) form as a consequence of simple local seemingly-chaotic interactions (e.g. local communication, stigmergy) [Hey99]. This kind of adaptiveness is also called **self-adaptiveness**, as it arises from the system itself without any external contributor.

Collective systems are especially complex, in fact in collective intelligence the connection between the individual behaviors of the system components and the collective behavior of the system is rarely straightforward. As a consequence, it may be difficult to design the individual components starting from the goal that the collective system should achieve. To tackle such complexity, one should adopt stricter and more formal approaches to software engineering, such as anticipating the verification of the system already during its design, using formal verification techniques such as **model checking** and **simulation**.

The applications of collective systems concern domains such as smart cities, complex sensor networks and the IoT [VAB+18], including pedestrian navigation (e.g. crowd evacuation), collective motion (e.g. drone fleet control [Vá14]) and pervasive IoT.

2.1.2 Aggregate Computing

Aggregate computing is an emerging paradigm for programming large-scale distributed situated systems, known as **aggregates** of **devices**, born to tackle the complexity of engineering such systems [VBD+19], including CAS.

The idea behind aggregate computing is to program the behavior of an aggregate directly at the *macro-level*, without explicitly programming the behavior of each of its individual components at the *micro-level*. In particular, a specification in aggregate computing defines how the components of an aggregate should behave and interact with each other in terms of how information propagates through the aggregate as a whole, moving the design focus from the individual to the collective. The propagation of information within an aggregate can be formally described using **field calculus**, which is the mathematical core of aggregate computing.

In field calculus, an aggregate is a network of devices capable of exchanging information between each other. The topology of the network (i.e. application-dependent physical or logical proximity of the devices) is described using a dynamic *neighboring relation*, which indicates the *neighbors* of each device (including the device itself), so that direct communication can only happen between a device and its neighbors. Information in the network is modelled at the *macro-level* as a

computational field, that is a function mapping each device to its corresponding state (or event) at a specific point in space and time. Finally, the propagation of information is a result of **functional composition**, **evolution** or **restriction** of computational fields.

The evolution of a computational field refers to its gradual transformation along the spatial or temporal dimensions. Evolution over space can be achieved with *inter-device communication*, including accumulation and elaboration of neighboring events for producing the next event of each device. Evolution over time can be obtained by specifying dependencies between the next and previous events of each device (potentially an expression of *intra-device communication*).

The restriction of a computational field refers to the application of a constraint on its evolution over space. Restriction can be achieved through *conditional partitioning of the network*, that is assigning each device to a different partition depending on a given condition, such that neighbors belonging to different partitions are isolated and cannot communicate despite their neighboring relation. However, only the restricted computational field is affected by the network partitions, so the information of other computation fields may still propagate between partitions.

More formally, a program specification in field calculus can be written using the abstract syntax in Figure 2.1. One such specification can be interpreted both at the *macro-level* (as a composition of operations on computational fields) and at the *micro-level* (as a composition of operations executed by each device every computation round to produce their next event). Such equivalent interpretations bridge the gap between the collective behavior of the aggregate and the individual behaviors of its components.

A **program** is a sequence of **function declarations** followed by an **expression** which determines the behavior of the system. An expression can be one of the following:

- A **variable**, referencing information (e.g. a function parameter).
- A **value**, expressing information. A value can be either a **local value** (e.g. a boolean, a number, any object) or a **neighboring value**, which is a function mapping, for each device, the neighbors to a local value.
- A **function call**, describing the composition of computational fields. The called function can be either **user defined** (i.e. referencing a function declaration) or **built-in** (e.g. arithmetic or logical operators).
- A **communication expression** `nbr`$\{e\}$, describing the evolution in space of a computational field. In detail, the expression yields a neighboring value computed in two steps: first each device computes the expression e sharing

$$
\begin{array}{lcllr l}
P & \Rightarrow & F^*e & & \textit{Program} & \\
F & \Rightarrow & \texttt{def}\ d(x^*)\{e\} & & \textit{Function Declaration} & \\
e & \Rightarrow & x & & \textit{Expression}: & \textit{Variable} \\
 & | & v & & : & \textit{Value} \\
 & | & f(e^*) & & : & \textit{Function Call} \\
 & | & \texttt{nbr}\{e\} & & : & \textit{Evolution over space} \\
 & | & \texttt{rep}(e)\{(x) \to e\} & & : & \textit{Evolution over time} \\
 & | & \texttt{if}(e)\{e\}\{e\} & & : & \textit{Restriction} \\
f & \Rightarrow & d & & \textit{Function Name}: & \textit{User-declared} \\
 & | & b & & : & \textit{Built-in} \\
v & \Rightarrow & l & & \textit{Value}: & \textit{Local Value} \\
 & | & \phi & & : & \textit{Neighboring Value} \\
l & \Rightarrow & \texttt{c}(l^*) & & \textit{Local Value}: & \textit{Constructor Call} \\
\phi & \Rightarrow & \delta^* \to l^* & & \textit{Neighboring Value}: & \textit{Devices} \to \textit{Local Values}
\end{array}
$$

Figure 2.1: An abstract syntax for field calculus [VBD+19]. The symbol a^* indicates a possibly empty sequence of a (e.g. $a_1, ..., a_n$ with $n \geq 0$), while the symbol $a^* \to b^*$ a possibly empty sequence of relations $a_1 \to b_1, ..., a_n \to b_n$. On the left, the production rules of the language. On the right, the meaning of the left and right side of each production rule.

the result with its neighbors; then each device collects the results of its neighbors producing a function mapping each neighbor to its latest evaluation of e.

- An **iteration expression** $\texttt{rep}(e_1)\{(x) \to e_2\}$, describing the evolution in time of a computational field. In detail, the expression is computed each round yielding a result v_i, with i being the number of rounds computed so far. The result v_0 computed in the first round is the value yielded by e_1, while successive results v_k ($k > 0$) are computed in the following rounds as the value yielded by e_2 when applying the function $s : (x) \to e_2$ to the result v_{k-1} of the previous round ($v_k = s(v_{k-1})$).

- A **branching expression** $\texttt{if}(e_1)\{e_2\}\{e_3\}$, describing the restriction of a computational field. In detail, the computation in the system is split depending on the condition e_1 (i.e. an expression evaluating to either true or false), resulting in the computation of e_2 where and when e_1 is satisfied or in the computation of e_3 otherwise.

In the branching expression, restriction happens as a consequence of **alignment**. Alignment is the process of keeping track of the structure of the field calculus specification (e.g. using an abstract syntax tree), in order to ensure correct message matching during communication when the specification contains different instances of `nbr` or `rep` constructs. Due to alignment, communication between a device and a neighbor can only happen if they are computing two expressions that share a `nbr` construct in the same position within the structure of the program. In particular, within the branching expression, alignment forbids communication between devices computing e_2 and e_3, as these expressions belong to two different branches of the program specification.

While field calculus provides solutions for the composition and evolution of global or regional behaviors in aggregates, its syntax is also too general for it to be resilient and too succinct for programming to be simple. Aggregate computing addresses the first problem by implementing three *resilient higher-order primitives* on top of field calculus (Listing 2.1).

```
def G(source, initial)(metric, accumulator){...}
def C(potential, local, null)(accumulator){...}
def T(initial, final)(decay){...}
```

Listing 2.1: The three higher-order primitives introduced by aggregate computing on top of field calculus.

The `Block G` primitive [VAB+18] handles the diffusion of information by computing the **gradient** (i.e. the computational field of distances) with respect to a `source`, while accumulating values towards the direction of increasing gradient. Accumulation starts from an `initial` value at the `source` and proceeds hop-by-hop using an `accumulator` moving away from the `source`. The distance between a device and its neighbors (used for computing the gradient) is defined by a `metric`.

The `Block C` primitive handles the convergence of information, using a `potential` (e.g. a gradient) to accumulate values towards the direction of decreasing `potential`. In this sense, the `Block C` primitive is complementary to the `Block G` primitive. Accumulation proceeds with each device applying an `accumulator` to a `null` value (idempotent for the `accumulator`), its `local` value and the values of any neighbor with a higher `potential`.

Finally, the `Block T` primitive handles the evolution of information in time, starting from an `initial` maximum value for each device and reducing it at each computation round using a `decay` function, until a `final` minimum value is reached.

These aggregate computing primitives cover the most common applications when programming aggregates, while offering additional resilience compared to field calculus due to their **self-stabilisation** property, which guarantees convergence towards a final stable state for the aggregate from any initial state after some time without changes in its environment. As such, they can be used as building blocks in *general or domain-specific libraries* for aggregate computing (e.g. swarm coordination framework [ACV23]), increasingly reducing the complexity of programming aggregates of devices.

2.1.3 Reactive Programming

Reactive programming is a paradigm built around the notions of *continuous time-varying values* and *propagation of change*, ideal for the development of event-driven applications [BCC+13]. In particular, computation is expressed in terms of dependencies between flows of information, so that when some information changes, all the dependent information is updated automatically by the underlying execution model.

Consider the following program for computing the sum of two variables.

```
var1 = 1
var2 = 2
var3 = var1 + var2  # var3: 3
var1 = 3            # var3: 5
var2 = 1            # var3: 4
```

In reactive programming, the program is translated into a **computational graph** (shown in Figure 2.2), expressing the dependencies of the variable `var3` on the variables `var1` and `var2`, so that any future reassignment of `var1` or `var2` will be automatically reflected on the value of `var3`, unlike standard imperative programming. Due to their non-standard behavior, variables in reactive programming are also called **reactive variables**.

A value assigned to a reactive variable can be either a **behavior**, that is a time-varying value in continuous time (e.g. time itself), or an **event stream**, that is a potentially infinite sequence of events, occurring at discrete points in time (e.g. mouse clicks). Generally, behaviors are used to model time-varying states, which can always be sampled, while event streams are used to model state updates, which exist only in the discrete point in time when they are triggered. However, some implementations of reactive programming avoid such distinctions.

In order to be applied to reactive variables, standard operators should be transformed into *reactive operators*. Such transformation is called **lifting** and requires changing the type signature of the operators and properly updating the computational graph. The semantics of reactive programming languages change depending on how lifting is implemented: *implicit lifting* allows to apply standard operators

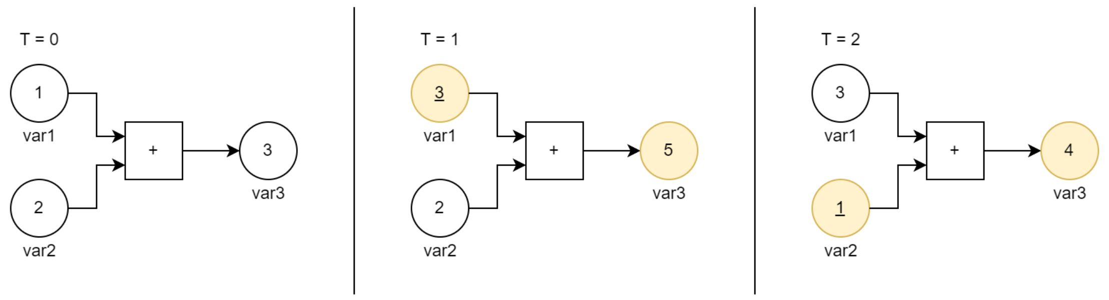

Figure 2.2: A computational graph in reactive programming: nodes (circles) represents reactive variables and their current values; yellow nodes represent a propagation of change; underlined yellow nodes represent the start of a propagation of change; operations (squares) represent a type of dependency between nodes (when the type of dependency is not relevant, they may be omitted). For example, at time `T=1`, `var1` was reassigned to value 3, triggering an update of `var3` to value 5.

to reactive variable as-is (transforming them under the hood); *explicit lifting* provides a **lift** primitive to apply the transformation to a standard operator; *manual lifting* does not implement lifting, requiring the developer to manually sample and compose the values of behaviors.

Reactive operators are used to build the computational graph of a reactive program, creating dependencies between reactive variables. Some reactive programming implementations enjoy the property of **multidirectionality**, allowing the definition of bidirectional dependencies or cyclic graphs. Some may support **switching**, allowing the definition of dynamic computational graphs whose dependencies change over time.

The **evaluation model** of a reactive programming language deals with the propagation of changes within a computational graph. Propagation of change always involves a **producer** to trigger the change (e.g. a dependency) and a **consumer** to react to the change (e.g. a dependent). The evaluation model can be categorized based on the roles of the two entities:

- **Pull-Based**: consumers poll producers for their events, resulting in *lazy reaction* (*demand-driven propagation*), as polling may happen after the time when the events were fired at the discretion of the consumer. This approach works best with time-varying values in continuous time.

- **Push-Based**: producers push events to the consumers, resulting in *eager reaction* (*data-driven propagation*), as state changes are propagated as they are produced. This approach works best when instantaneous reactions are a requirement.

While the push-based evaluation model is adopted in most recent implementations of reactive programming, it requires additional mechanisms to avoid **glitches**, which are inconsistent events, generated when a dependent is updated before all of its dependencies are up-to-date, resulting in a combination of new and stale events. Consider the following example.

```
1  var1 = 1
2  var2 = var1 * 1     # var2: 1
3  var3 = var1 + var2  # var3: 2
4  var1 = 2            # var3: 3 (glitch); var2: 2; var3: 4 (correct)
```

In the example, when `var1` is reassigned (line 4), the propagation of change may reach `var3` before `var2`, leading to an inconsistent value for `var3`, since `var2` is not up-to-date. Eventually, `var2` will also be updated and so `var3` will reach a consistent value. However, any dependency on `var3` would have already suffered from its inconsistencies (e.g. incorrect program state, wasteful recomputations), hence the requirement of mechanisms for **glitch freedom**. Note that glitches are a consequence of inconsistent individual handling of simultaneous events or reactions.

Most implementations of push-based reactive programming guarantee glitch freedom in non-distributed environments. However, an important extension of reactive programming is **distributed reactive programming**, which allows expressing and managing the dependencies between the components of a distributed system by distributing the nodes of a computational graph across multiple machines (e.g. in "`var3 = var1 + var2`", `var1`, `var2` and `var3` may be located in different machines). Recent progress shows that is possible to guarantee glitch freedom also in push-based distributed reactive programming for acyclic graphs [MSM19], or at different levels of consistency [MS18], while retaining scalability and parallelism.

Reactive programming is most suitable for designing event-driven applications, achieving better declarativity and looser coupling between components with respect to standard **event-driven programming** paradigms, such as the *observer* pattern[1]. In fact, the former hides how the propagation of change is implemented in the system, letting the developer focus solely on the behavior of the program, while the latter requires the developer to manually implement dependencies as events that may trigger dependent events, resulting in a flow of control that is harder to understand and nested transitive dependencies that are harder to detect.

[1]a pattern for event-driven programming, in which consumers react to events by registering some callbacks (*listeners*) to the event producers, so that they may be executed each time a new event is triggered. Callbacks may also trigger other events, creating a dependency graph between callbacks. In fact, most reactive implementations are an abstraction over this pattern.

2.1.4 Functional Reactive Programming

Functional Reactive Programming (FRP) is a subset of both reactive programming and **functional programming**, retaining the advantages of reactive programming, while promoting **compositionality**, which is a property of semantics, holding if the meaning of an expression is solely determined by the meaning of its parts and the rules used to combine them [BJ16].

In functional programming, compositionality is achieved by expressing software behaviors as **pure functions**, that is functions in the mathematical sense of the term. Pure functions produce no observable side effects when applied and are **referentially transparent**, meaning that different applications of a function to the same input always produce the same outputs. To attain referential transparence, functions should avoid referencing *shared mutable data* so that their behavior is constant since their definition and has no side effects.

In reactive programming, compositionality also requires glitch freedom, as observable glitches may invalidate the behavior expressed by a function (e.g. the behavior of a function may change due to inconsistent handling of simultaneous events by the underlying evaluation model).

Compositionality is essential for dealing with complex software, tackling its complexity by composition of simpler components that are easier to reason about. Moreover, it deals with scalable software, tackling their growing complexity over time by facilitating the addition of new features to existing composable applications.

2.2 Technologies

This section provides an overview of the specific technologies referenced by this project, in relation to the general concepts described in the previous section.

2.2.1 Sodium

Sodium[2] is a BSD-licensed library implementing FRP in several languages (including Java), inspired by many previous implementations of FRP. Sodium is meant to be a *true* FRP implementation, in the sense that it provides full compositionality compared to other implementations (e.g. Reactive Extensions (Rx)[3], which is not glitch-free) [BJ16].

In Sodium, behaviors are modelled as **cells** with denotation `Cell[V]`, indicating a time-varying value of type `V`, while event streams are modelled as simply **streams**

[2]`https://github.com/SodiumFRP`
[3]`https://github.com/ReactiveX`

with denotation `Stream[E]`, indicating a sequence of emissions of events of type `E`. In particular, a `Stream` is defined as a list of events bound to the time when they were fired, while a `Cell` is defined as a pair of its initial value together with a `Stream` of its updates over time.

Time is represented as a sequence of **transactions**, which can be interpreted as atomic time units. Only one transaction at a time can be executed by the engine of Sodium, even when considering multiple independent computational graphs. During a transaction, first all events are processed simultaneously keeping all values constant (i.e. immutable **transactional context**), then all time-varying values are updated accordingly. A transaction is started automatically each time an event is pushed in the computational graph and closed only after its corresponding propagation of change has been completed. Alternatively, it is possible to create a new transaction explicitly using the `Transaction.run` method (e.g. useful for sending simultaneous events, graph initialization or handling forward references).

Sodium provides a set of built-in core primitives for building static acyclic computational graphs (Listing 2.2), creating and combining `Cell`s and `Stream`s. These include:

- `never`: create a new `Stream` that will never emit events.
- `map`: given a `Stream` s in input, create a new `Stream` s' whose events are the events of s transformed with a given `mapping` function. An analogous operation is provided for `Cell`s.
- `filter`: given a `Stream` s in input, create a new `Stream` s' whose events are the events of s, discarding those which do not satisfy a given `predicate`.
- `merge`: given two `Stream`s s_1 and s_2, create a new `Stream` s' whose events are the events fired by either s_1 or s_2, combining their simultaneous events with a given `merging` function.
- `snapshot`: given a `Stream` s and a `Cell` c, create a new `Stream` s' whose events are the events of s combined with the most recent value of c using a given `combine` function.
- `constant`: create a `Cell` c holding a given `value` forever.
- `hold`: given a `Stream` s, create a `Cell` c holding a given `initial` value, which is updated each time s fires a new event. In particular, s is the `Stream` of updates of c.
- `sample`: given a `Cell` c, obtain its most recent value. This primitive should not be used when mapping or lifting `Cell`s as it would break referential

transparence, which is preserved for other primitives by exploiting the immutability of transactional contexts.

- `lift`: given two `Cells` c_1 and c_2, create a new `Cell` c whose value is obtained by combining the values of c_1 and c_2 using a given `operator`. In particular, the value of c is updated each time the values of c_1 or c_2 are updated. Note that lifting is explicit in Sodium.

```
type Stream[E]
type Cell[V]

def never[E]: Stream[E]
def map[A, B](s: Stream[A], mapping: A => B): Stream[B]
def filter[E](s: Stream[E], predicate: E => Boolean): Stream[E]
def merge[E](s1: Stream[E], s2: Stream[E], merging: (E, E) => E): Stream[E]
def snapshot[A, B, C](s: Stream[A], c: Cell[B], combine: (A, B) => C): Stream[C]

def constant[V](value: V): Cell[V]
def hold[V](s: Stream[V], initial: V): Cell[V]
def sample[V](c: Cell[V]): V
def map[A, B](c: Cell[A], mapping: A => B): Cell[B]
def lift[A, B, C](c1: Cell[A], c2: Cell[B], operator: (A, B) => C): Cell[C]
```

Listing 2.2: An abstract view on the Sodium primitives for constructing static acyclic computational graphs. Some primitives can be derived as a combination of the others.

Sodium also provides support for dynamic computational graphs, including graph expansion, reduction and more general sub-graph substitution. A dynamic computational graph can be represented as a time-varying computational graph, that is a `Cell` holding a reactive variable as value (either other `Cells` or `Streams`). In particular, two switching operators are implemented in Sodium (Listing 2.3): `switchS` builds a dynamic computational graph from a `Cell` of `Streams` cs, creating a new `Stream` s' whose events are the events of the most recent `Stream` held by cs; `switchC` works similarly for `Cell` of `Cells`.

```
def switchS[E](cs: Cell[Stream[E]]): Stream[E]
def switchC[V](cc: Cell[Cell[V]]): Cell[V]
```

Listing 2.3: An abstract view on the Sodium primitives for constructing dynamic computational graphs.

Support is also provided for cyclic computational graphs. However, since a node declares its dependencies on other defined nodes during its creation, cyclic dependencies are not possible without a mechanism for forward referencing, allowing a node to declare a dependency on another node that is yet to be defined (e.g. itself). Sodium allows forward referencing in Java by decoupling the declaration and definition of a node using the type `CellLoop[V]` (or `StreamLoop[E]`), which is used for declaring a node that will be assigned later to a defined `Cell` (or Stream) through its method loop. In other words, CellLoop acts as a placeholder, referencing a Cell that is not yet available. Still, declaration and definition should happen conceptually at the same time to avoid the propagation of change to empty references, hence a CellLoop must be declared and assigned within the same transaction.

```
type StreamLoop[E] <: Stream[E]
type CellLoop[E] <: Cell[E]
def streamLoop[E]: StreamLoop[E]
def loop[E](reference: StreamLoop[E], value: Stream[E]): Stream[E]
def cellLoop[E]: CellLoop[E]
def loop[V](reference: CellLoop[V], value: Cell[V]): Stream[V]
```

Listing 2.4: An abstract view on the Sodium primitives for constructing cyclic computational graphs.

Interoperability with non-FRP software interfaces is provided via a set of **operational primitives** (Listing 2.5), which are excluded from the core primitives since their incorrect usage may break some properties of Sodium. A broker between an FRP interface and a non-FRP interface can be implemented using the type CellSink[V] (or StreamSink[E]), which is a Cell (or Stream) that supports event pushing. In particular, the send primitive implements non-FRP to FRP interactions, allowing to push an update to a CellSink and managing the propagation of change through a push-based evaluation model (i.e. the caller of send will update all the dependent nodes in the computation graph). Conversely, the listen primitive implements FRP to non-FRP interactions, allowing to register a callback to execute any time the state of a Cell is updated (such subscription can be cancelled using the returned Listener). Note that using send within a callback is not allowed, as it could be used to implement custom primitives that violate compositionality. For the same reasons, Sodium discourages and forbids inheritance of its types. Instead, custom primitives should be implemented as a combination of the core primitives to preserve compositionality.

```
type StreamSink[E] <: Stream[E]
type CellSink[V] <: Cell[V]
def send[E](s: StreamSink[E], event: E): Unit
def send[V](c: CellSink[V], update: V): Unit
def listen[E](s: Stream[E], callback: E => Unit): Listener
def listen[V](s: Cell[V], callback: V => Unit): Listener
```

Listing 2.5: An abstract view on the Sodium operational primitives.

Additionally, Sodium offers other operational operators to tackle some specific practical problems (e.g. value, updates, split, defer...) and many more helper primitives to facilitate the construction of computational graphs (e.g. accum, collect, sequence...). While these operators won't be discussed here, since they are not as relevant for this project, more information about them and Sodium can be found in the book [BJ16]. The book also describes some useful FRP patterns, such as the **calming** pattern, useful to create **calm** reactive variables, which avoid firing consecutive repetitions of the same event, reducing redundant re-computations.

The authors compare other standard event-programming paradigms (specifically the observer pattern) to Sodium, highlighting several bugs that are common in the former, which are banished in the latter if used as intended. In particular, Sodium promises to solve the following problems:

- *Unpredictable order*: in complex networks of callbacks, it is difficult to track the order in which they are executed. Sodium abstracts over event ordering making it completely undetectable.
- *Missed first event*: it is difficult to guarantee that callbacks are registered before the first event. Sodium can solve the problem by initializing the program within a transaction.
- *Messy state*: callbacks tend to describe behaviors as state machines, which are difficult to maintain. Sodium solves the problem using the declarativity of the FRP paradigm.
- *Threading issues*: executing callbacks in parallel may lead to deadlock due to synchronization. Sodium solves the problem by executing only one transaction at a time.
- *Leaking callbacks*: forgetting to deregister a callback from a producer causes memory leaks and wastes CPU time. Sodium automatically deregisters callbacks that are not used any longer.
- *Accidental recursion*: it is easy to introduce accidental cyclic dependencies between nested callbacks. Sodium solves the problem using the declarativity of the FRP paradigm.

In addition, Sodium grants the compositionality required to tackle the complexity of complex systems.

2.2.2 ScaFi

Scala Fields (ScaFi)[4] is an open-source aggregate computing framework for the Scala programming language, providing a usable internal Domain Specific Language (DSL) for aggregate specifications and a platform for the simulation and execution of such specifications [CAV].

[4]`https://github.com/scafi`

In ScaFi, the core concepts of field calculus are modelled by a **trait** (i.e. an interface) like the one reported in Listing 2.7 [VBD+19], whose methods represent the constructs of field calculus.

```
trait FieldCalculus:
  def nbr[E](exp: => E): E
  def rep[E](exp: => E)(evolve: E => E): E
  def foldhood[E](exp: => E)(accumulate: (E, E) => E)(nbrExp: => E): E
  def aggregate[E](exp: => E): E

  // platform interactions
  def mid: Id
  def sense[V](name: String): V
  def nbrvar[V](name: String): V
```

Listing 2.6: The core constructs of field calculus, represented as a trait, abstracting over the actual organization within ScaFi.

ScaFi provides no explicit reification for computational fields. Indeed, any Scala expression is treated implicitly as a field calculus expression, yielding a computational field. For instance, the expression "1 + 2" yields constant uniform computational field holding the value 3 at any point in space and time, obtained as the point-wise summation of a field of "1"s and a field of "2"s (Figure 2.3).

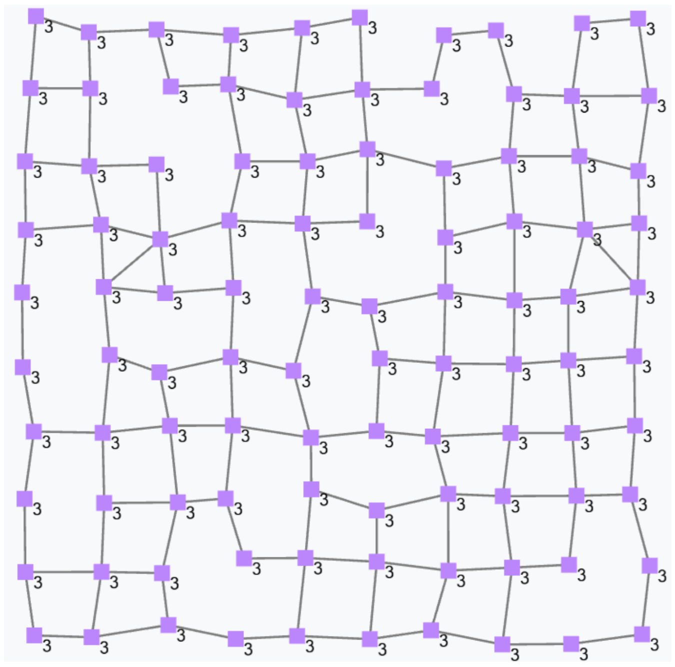

Figure 2.3: A graph representing an aggregate of devices (*nodes*) and their neighboring relations (*edges*). In particular, it represents the computational field yielded by the expression "1 + 2" [CAV].

Despite being equivalent, the semantics of ScaFi differ from the semantics of field calculus for some operators: evolution over space is implemented with a combination of the `nbr` and `foldhood` operators, the latter exploiting the former to accumulate the values of neighbors in each device (i.e. `nbr` does not yield a neighboring value directly as in field calculus); restriction is implemented using the `aggregate` operators, which handles selective partitioning; evolution over time with `rep` follows the same semantics as field calculus.

Additionally, ScaFi provides contextual operators that handle interactions with the underlying platform, namely `mid`, which computes the field of the device identifiers in the aggregate, `sense`, which computes a field of the values perceived by a specific sensor from the environment (e.g. a field of temperatures), and `nbrvar`, which computes a field mapping each neighbor to a value perceived by a specific sensor from the environment (e.g. a field of distances with each neighbor).

The core DSL can be extended with **mixins** to provide higher-level primitives and operators. ScaFi already includes some built-in extensions, such as the resilient aggregate computing blocks (Listing **??**).

```scala
trait AggregateComputing:
  self: FieldCalculus =>
  // aggregate computing blocks
  def G[V](source: Boolean, initial: V, accum: V => V, metric: () => Double): V
  def C[P: Bounded, V](potential: P, accum: (V, V) => V, local: V, nullV: V): V
  def T[V: Numeric](initial: V, floor: V, decay: V => V): V
  def S(grain: Double, metric: () => Double): Boolean

  // derived operators
  def branch[E](cond: => E)(th: => E)(el: => E): E
  def mux[E](cond: => E)(th: => E)(el: => E): E
  def share[E](exp: => E)(evolve: (E, () => E) => E): E
```

Listing 2.7: The core constructs of aggregate computing, represented as a mixin for field calculus, abstracting the actual organization within ScaFi.

The higher-level primitives in ScaFi include but are not limited to the already presented `G`, `C` and `T` blocks of aggregate computing, an additional `S` block, which handles sparse leader election based on proximity, a `branch` operator, implementing the branching expression of field calculus (relying on `aggregate`)[5], and a new `share` operator, which handles the evolution over time of a neighboring value (indeed a combination of the behaviors of `rep` and `nbr` in field calculus, albeit much more efficient [ABD$^+$19]).

The execution of a ScaFi specification is performed by the underlying platform, which adopts an *asynchronous* **round-based** execution model, in which a round is the computation required for an individual device to produce its next output

[5]conditional computation without partitioning is implemented by the `mux` operator instead, which is equivalent to an *if-then-else* expression in Scala.

based on the aggregate specification. A round consists of the following three steps in order:

1. **sense**: the device updates its current **context** (i.e. all known information in its perspective), by retrieving its previous output, the information perceived through its *sensors* from the local environment and the messages arriving from neighboring devices;

2. **compute**: the device computes its current output by executing the aggregate specification against its current context. The output of a device is an abstract syntax tree, tracking the structure of the executed aggregate specification for alignment. In particular, the root of the tree contains the final result of the computation, while the roots of its sub-trees contain the results of sub-computations.

3. **interact**: the device broadcasts some information extracted from its output (called an **export**) to neighboring devices and updates the local environment through its *actuators*. The export can be derived from the output of the device by searching in the abstract syntax tree for operations involving communication (e.g. sub-trees depending on **nbr**).

Support for simulation is also implemented by several ScaFi modules or through integration with third-party simulators (e.g. Alchemist[6] [PMV13]).

2.2.3 FRASP

FRASP (Functional Reactive Approach to Self-organisation Programming)[7] is a new open-source aggregate computing framework for the Scala programming language, currently under active research.

FRASP draws inspiration from ScaFi, sharing many similarities. The key distinction lies in the implemented execution model: the former adopts a novel functional reactive execution model, leveraging the Sodium library, as opposed to the round-based execution model of the latter, common in aggregate computing [CDA+23].

The motivation behind FRASP is to provide for some of the shortcomings of the round-based execution model, including *periodic computation*, *complete re-computation* and *redundant message exchanges*. Indeed, the benefits of adopting the execution model of FRASP for aggregate computing are the following:

[6]`https://github.com/AlchemistSimulator/Alchemist`

[7]`https://github.com/cric96/distributed-frp`

- *Event-driven computation*: in a device, computation is driven by relevant changes in its perception of the environment (e.g. sensors, neighbor data). In other words, computation is performed only when required.
- *Independent scheduling of sub-computations*: when a device detects a change in its context, only the dependent sub-computations of its programs are re-computed. In other words, full re-computations of an aggregate specification are avoided when possible.
- *Minimal communication*: a device only broadcasts its exports upon relevant changes, avoiding further message exchanges after the aggregate reaches a stable configuration. In other words, redundant computation caused by repeated messages is avoided.

In FRASP, computational fields are reified into Sodium's `Cells`, which neatly capture their time-varying nature. Like FRP, a specification is the configuration of a computational graph, which tracks the dependencies between computational fields and manages the propagation of change automatically.

Computational fields are initialized by `Flows`, which model sub-computations in an aggregate specification and are first-class citizens in FRASP. The purpose of `Flows` is to defer the construction of the computational graph until the devices of the aggregate network are initialized, which is required to express dependencies related to their neighbors and sensors. In addition, `Flows` also keep track of their position inside the FRASP specification, building the abstract syntax tree used for alignment.

The semantics of FRASP (Listing 2.8) faithfully resembles the semantics of field calculus, while also sharing common constructs with ScaFi. However, since computational fields have been reified, additional operators are required to adapt values yielded by plain Scala expressions to the language constructs, namely `constant` for values and `lift` for operators (*lifting*).

The main difference with the field calculus semantics is the `loop` construct, replacing the `rep` construct. The `loop` construct implements the evolution of a computational field over time as a (cyclic) self-dependency within the computational graph of a FRASP specification, rather than relying on the concept of computation round. Indeed, the previous state of a device is computed through self-alignment, leveraging the fact that every device is a neighbor of itself.

```scala
trait FraspLanguage:
  // field calculus
  type Flow[V]
  def constant[V](value: V): Flow[V]
  def lift[A, B, C](a: Flow[A], b: Flow[B])(operator: (A, B) => C): Flow[C]
  def loop[V](init: V)(evolve: Flow[V] => Flow[V]): Flow[V]
  def nbr[V](cond: Flow[Boolean])(th: Flow[V])(el: Flow[V])
    : Flow[NeighboringValue[V]]
  def branch[V](cond: Flow[Boolean])(th: Flow[V])(el: Flow[V]): Flow[V]

  // platform interactions
  def mid: Flow[DeviceId]
  def sensor[V](name: LocalSensorId): Flow[V]
  def nbrSensor[V](name: NeighborSensorId): Flow[NeighboringValue[V]]

  // derived operations
  def mux[V](cond: Flow[Boolean])(th: Flow[V])(el: Flow[V]): Flow[V]
  def share[V](init: Flow[V])(evolve: Flow[NeighboringValue[V]] => Flow[V])
    : Flow[V]
```

Listing 2.8: The core constructs of the FRASP language, represented as a trait, abstracting over the actual organization within FRASP.

FRASP also provides a basic simulator implementing its reactive execution model (Figures 2.4 and 2.5). On an abstract level, the simulator operates in two phases:

- **Configuration**: accept a FRASP specification, which describes the configuration of a computational graph, and an environment, which describes the devices of the aggregate and their neighboring relations (e.g. based on proximity);

- **Execution**: create the devices and build the computational graph of the aggregate, based on the FRASP specification. In doing so, the simulator establishes the dependency chains from the perceptions of each device (e.g. neighbor and environmental data) to its exports and from its exports to the neighbor data perceived by its neighbors. As soon as the graph is built, the *input nodes*[8] of the computational graph will propagate their initial value to all their dependents, then the computation is carried on automatically by the underlying FRP engine indefinitely.

  Since non-trivial specifications for aggregate computing include cyclic dependencies in the computational graph, additional measures must be taken to avoid the indefinite propagation of non-relevant changes (e.g. redundant messages). In particular, FRASP applies the FRP calming pattern to all nodes when building a computational graph, allowing self-stabilizing specifications to eventually reach a stable state, in which events are no longer propagated in the aggregate until the next change in the environment. Note that the execution is still indefinite even after reaching a stable state, since it is always possible for an event to happen in the future, however, no propagation of changes implies no consumption of computational resources (i.e. the aggregate keeps waiting for an event to occur).

[8]a node initialized by a leaf `Flow` in the abstract syntax tree of a FRASP specification: either `constant`, `mid`, `sensor`, `nbrSensor` or `loop`, as they do not require other `Flow`s in input.

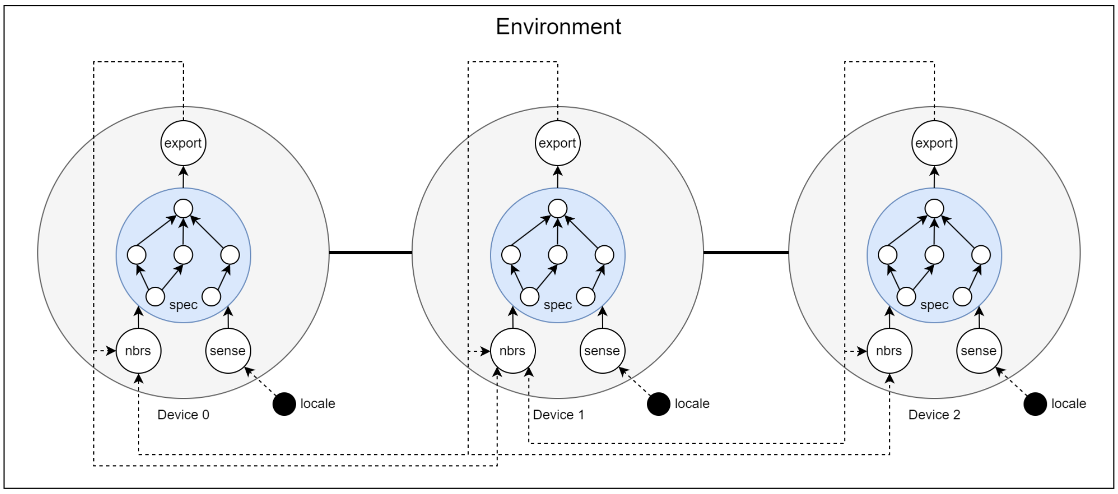

Figure 2.4: The reactive execution model of FRASP. In the diagram, there are three devices (*grey circles*), each configured with an aggregate specification (*blue circles*). For each device, the input of the aggregate specification is neighboring (*nbrs*) and local environmental data (*sense*); the output is the export that needs to be broadcast to neighbors (*export*). The figure also shows internal (*solid arrow*) and external (*dashed arrow*) dependencies in the computational graph.

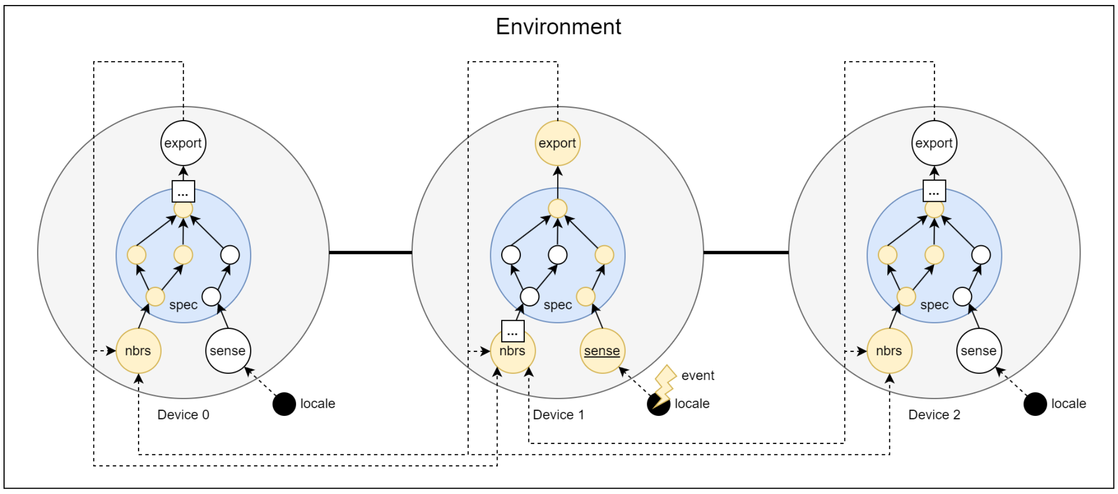

Figure 2.5: An example of propagation of change in the execution model of FRASP. The local environment of device 1 changed, causing changes to all its dependents. The three dots indicate that the change would continue to propagate afterward following the graph dependencies. Note how the propagation of change would carry on indefinitely in any cyclic graph without proper measures.

Since this project contributes to the implementation of FRASP, a brief overview of its architecture is due. Internally, FRASP is organized into the following three layered modules (Figure 2.6):

- `frp`: provide extensions and abstractions over the FRP engine on which the framework depends.
- `core`: provide the model and implementation of the FRASP specification, as illustrated previously in Listing 2.8.
- `simulation`: provide a basic simulator for running aggregate specifications over a network of devices.

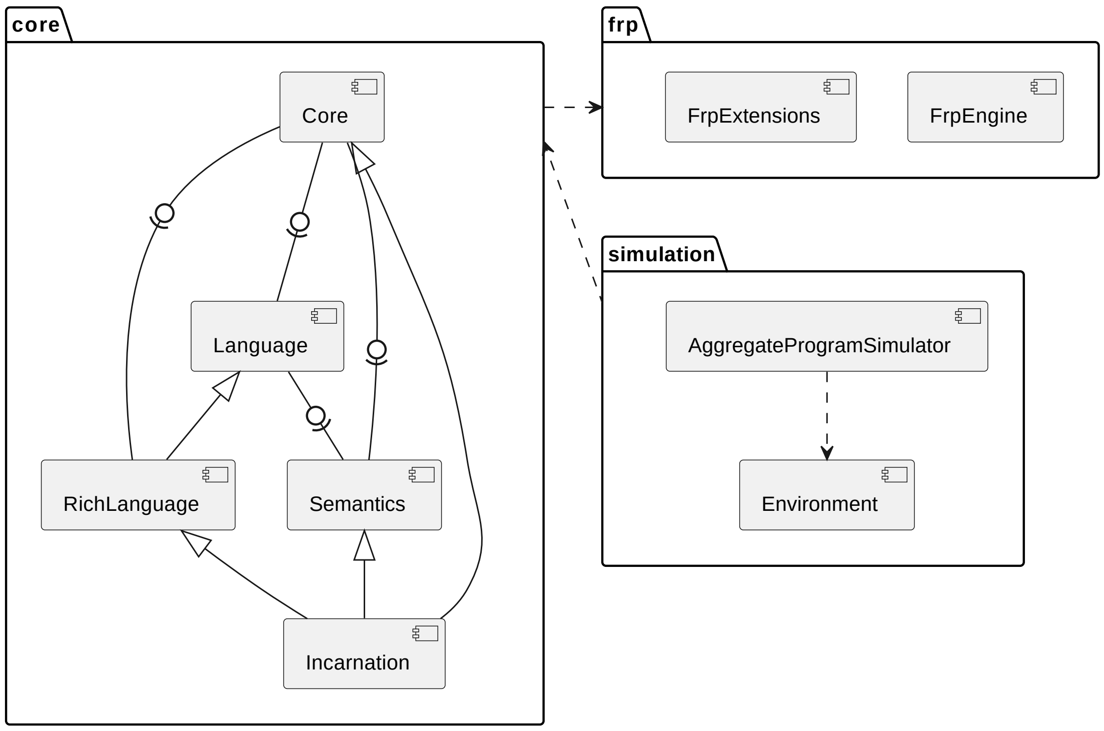

Figure 2.6: The architecture of FRASP [CDA+23].

The contributions of this project concern mostly the `frp` and `simulation` modules. More details will be provided in the following chapters as needed.

Chapter 3

Analysis

Jahrim Gabriele Cesario: What is the problem that needs to be solved? Requirements...

Chapter 4

Design

Jahrim Gabriele Cesario: How was the problem solved?

Chapter 5

Implementation

Jahrim Gabriele Cesario: How was the solution implemented concretely?

Chapter 6

Evaluation

Jahrim Gabriele Cesario: How was the solution evaluated and tested?

Chapter 7

Conclusions

Jahrim Gabriele Cesario: Brief summary What has been achieved? What has not been achieved? What are future explorations for the work done in this project?

Bibliography

[ABD$^+$19] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, Danilo Pianini, and Mirko Viroli. Field-based coordination with the share operator. *Log. Methods Comput. Sci.*, 16, 2019.

[ACV23] Gianluca Aguzzi, Roberto Casadei, and Mirko Viroli. Macroswarm: A field-based compositional framework for swarm programming. In Sung-Shik Jongmans and Antónia Lopes, editors, *Coordination Models and Languages*, pages 31–51, Cham, 2023. Springer Nature Switzerland.

[BCC$^+$13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4), aug 2013.

[BJ16] Stephen Blackheath and Anthony Jones. *Functional Reactive Programming.* Manning, July 2016.

[CAV] Roberto Casadei, Gianluca Aguzzi, and Mirko Viroli. Scafi documentation. `https://scafi.github.io/`. Accessed: 02-07-2024.

[CDA$^+$23] Roberto Casadei, Francesco Dente, Gianluca Aguzzi, Danilo Pianini, and Mirko Viroli. Self-organisation programming: A functional reactive macro approach. In *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 87–96, 2023.

[Cen] Scala Center. Scala documentation. `https://docs.scala-lang.org/`. Accessed: 02-08-2024.

[Fer15] Alois Ferscha. Collective adaptive systems. In *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*, UbiComp/ISWC'15 Adjunct, page 893–895, New York, NY, USA, 2015. Association for Computing Machinery.

[Hey99] Francis Heylighen. The science of self-organization and adaptivity. *Center "Leo Apostel", Free University of Brussels, Belgium*, 1999.

[LP00] Yang Liu and Kevin M. Passino. Swarm intelligence: Literature overview. *Department of Electrical Engineering, The Ohio State University, Ohio*, March 2000.

[MS18] Alessandro Margara and Guido Salvaneschi. On the semantics of distributed reactive programming: The cost of consistency. *IEEE Transactions on Software Engineering*, 44:689–711, 2018.

[MSM19] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Distributed reactive programming for reactive distributed systems. *CoRR*, abs/1902.00524, 2019.

[Ora] Oracle. Java documentation. `https://dev.java/`. Accessed: 02-08-2024.

[PMV13] D Pianini, S Montagna, and M Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *Journal of Simulation*, 7(3):202–215, August 2013.

[VAB$^+$18] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Trans. Model. Comput. Simul.*, 28(2), mar 2018.

[VBD$^+$19] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *Journal of Logical and Algebraic Methods in Programming*, 109:100486, 2019.

[Vá14] Dr. Gábor Vásárhely. The world's first autonomous outdoor quadcopter flock. `https://hal.elte.hu/~vasarhelyi/en/projects/ercdrones/`, 2014. Accessed: 02-02-2024.