ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

Master Degree in Computer Science and Engineering

Event-driven Simulation and Verification of FRASP Systems Against Spatio-Temporal Properties

Master thesis in:
PERVASIVE COMPUTING

Supervisor
Prof. Mirko Viroli

Cosupervisors
Dr. Roberto Casadei
Dr. Gianluca Aguzzi

Candidate
Jahrim Gabriele Cesario

IV Graduation Session
Academic year 2022-2023

Abstract

The growing relevance of large-scale distributed systems, including collective adaptive systems, has inspired the research for novel aggregate computing paradigms, addressing the complexity of programming macro-level behaviors over sizeable networks of devices. One of the most recent advancements in this direction is the development of a reactive execution model for such systems, embodied in a domain-specific language (DSL) called FRASP (Functional Reactive Approach to Self-organization Programming), which overcomes several limitations of the previous round-based execution models, such as redundant re-computations.

The objective of this thesis is to consolidate FRASP by developing an extensive test suite, demonstrating the correctness of the current implementation, supporting future extensions of the language, and providing valuable insights into the implications of the reactive execution model and the challenges to overcome. With this goal in mind, the event-driven simulator provided by FRASP has been re-designed to enable the evaluation of spatio-temporal properties on the evolution of aggregate systems, focusing on the convergence of self-stabilizing specifications towards an expected stable state.

In conclusion, the test suite verifies the overall correctness of FRASP, except for a couple of issues concerning the implementation, which have been identified and analyzed, so that they may be addressed in future works.

To my family, my friends, and my partner,
thanks for your support and patience

Acknowledgements

I would like to start by expressing my gratitude towards the people who supported me during this project.

Many thanks to,

my supervisor, Prof. Mirko Viroli, and co-supervisors, Dr. Roberto Casadei and Dr. Gianluca Aguzzi, for giving me the opportunity to work on this thesis, supporting me during its development. I am especially thankful for their generosity in sharing their knowledge with me.

my family and my partner, for their compassion and comfort during these challenging past months. A special thanks to my partner for dedicating her time to review my work and providing valuable feedback.

my friends, for their understanding and their patience, during a time when I have not been as present. Hopefully, this will change in the future.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Motivation and Goals

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], whose specifications directly describe robust collective behaviors for networks of devices.

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 (Functional Reactive Approach to Self-organization Programming) language [CDA$^+$23], which is the subject of this work.

FRASP introduces a novel Domain Specific Language (DSL) that combines the functional reactive programming paradigm with the aggregate computing paradigm. This extension allows the application of the former in distributed systems, specifically in collective adaptive systems. Additionally, it improves the latter with an optimized 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 language requires a consolidated test suite before further evolution, to assess the correctness of its current implementation, prevent possible software regressions, that may be due to unsuspected interactions between present and future features, and possibly discover unforeseen implications of the reactive model.

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

1.2 Structure

The content of this 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. Then, Chapter 3 analyzes the objectives and requirements of this thesis, defining an outline for the strategy to adopt. Afterwards, Chapter 4 describes the solution designed for the project and Chapter 5 delves into the details of its concrete implementation. Towards the end, Chapter 6 explains the verification methods applied to the implemented solution, analyzing their results. Finally, Chapter 7 provides a summary of the achievements and future explorations of this project.

1.3 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 pseudocode, due to its clean and minimalistic functional syntax.

1.4 Artifacts

The documented source code for this project is available in a public GitHub repository[1], which can be downloaded to supplement the content of this document, providing detailed insights into the implementation.

[1]`https://github.com/jahrim/distributed-frp/tree/test/functional-test-suite`

Chapter 2

Background

2.1 Concepts

This section provides a general technology-agnostic description of the main concepts referenced by this project, namely collective adaptive systems, aggregate computing, reactive programming, and functional reactive programming.

2.1.1 Collective Adaptive Systems

Collective systems are 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**, in fact it emerges from the behaviors of its individual components, the local interactions between them and with their environment [Cas23a]. 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** [LP00].

Adaptiveness is the ability of a system to change its behavior depending on the circumstances to better achieve its goals. This property is essential in the applications of distributed collective systems, to the extent that they are often called directly **Collective Adaptive Systems (CASs)**. In fact, adaptiveness grants CASs 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 inherently 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 [BPV15], including CAS.

The idea behind aggregate computing is **macroprogramming** [Cas23b], that is programming the behavior of the aggregate directly at the *macro-level*, without explicitly defining 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 (Figure 2.1).

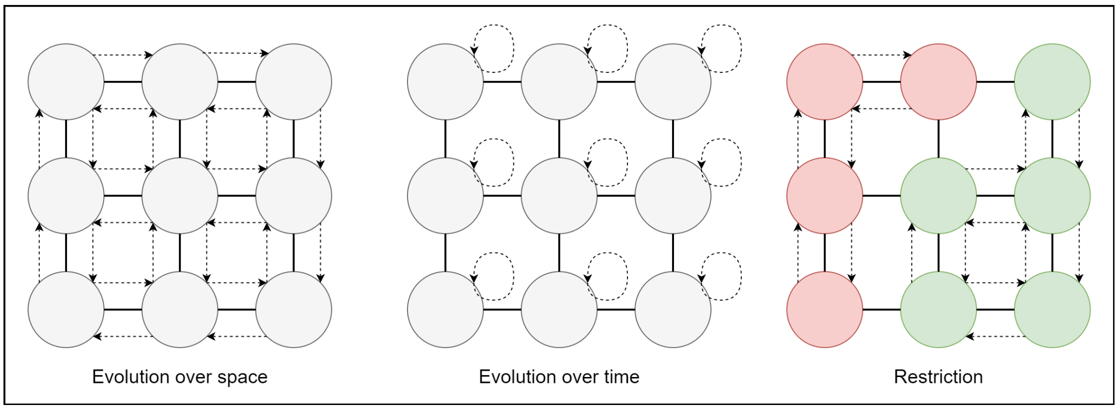


Figure 2.1: The three methods of propagation of information in field calculus. The figure shows three aggregates, each containing nine devices (*circles*), connected by neighboring relations (*solid lines*). Each aggregate shows the propagation of information (*dashed arrows*) using one of the methods of field calculus. Composition of these methods is also possible.

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* or *self-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 computational fields may still propagate between partitions.

More formally, a program specification in field calculus can be written using the abstract syntax in Figure 2.2. One such specification can be interpreted both at the *macro-level* (as a composition of collective operations on computational fields) and at the *micro-level* (as a composition of individual operations executed by every device to compute and propagate its next event). Such equivalent interpretations bridge the gap between the collective behavior of the aggregate and the individual behaviors of its components.

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

Figure 2.2: 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 $a^*{\rightarrow}b^*$ a possibly empty sequence of relations $a_1{\rightarrow}b_1, ..., a_n{\rightarrow}b_n$. On the left, the production rules of the language. On the right, the meaning of the left and right side of each production rule.

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

- 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 over 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

the result with its neighbors; then, each device collects the results of its neighbors, producing a neighboring value of the latest evaluation of e.

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

- A **branching expression** `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 a 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 problems by implementing three *resilient higher-order primitives* on top of field calculus (Listing 2.1 and Figure 2.3).

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`

```
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.

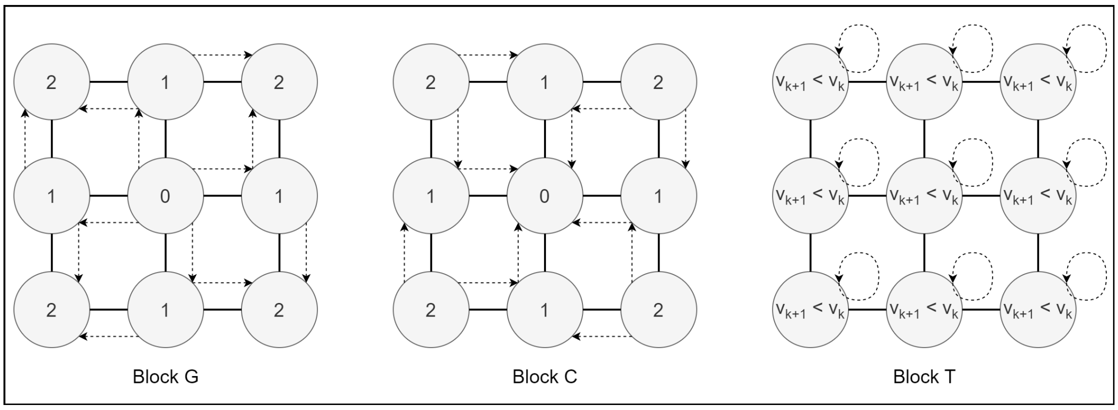


Figure 2.3: The three resilient methods of propagation of information in aggregate computing: `Block G` propagates information towards the direction of increasing gradient (*value inside each device*), diffusing information; `Block C` propagates information towards the direction of decreasing gradient, collecting information; `Block T` evolves information in time, applying a decay until convergence to a minimum value. Note that the diagrams only shows *relevant* propagation of information, hiding the underlying communication required to attain such behavior.

`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-stabilization** property, which guarantees convergence towards a final stable state for the aggregate after some time without changes in its environment or its network topology. 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 2.4), 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**.

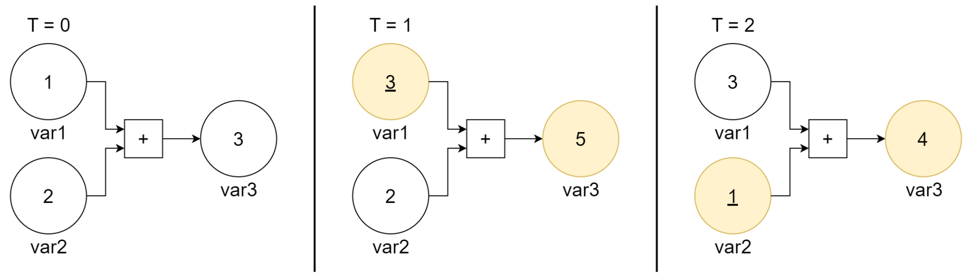


Figure 2.4: A computational graph in reactive programming: nodes (*circles*) represent reactive variables and their current values; yellow nodes represent a propagation of change; underlined yellow nodes represent the start of a propagation of change; operations (*squares*) represent a type of dependency between nodes (when the types of dependency are 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.

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). Typically, 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 changes depending on how lifting is implemented: *implicit lifting* allows applying standard operators to reactive variables 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 possess the property of **multi-directionality**, 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 a change (i.e., a dependency) and a **consumer** to react to the change (i.e., 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* (or *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* (or *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.

```
var1 = 1
var2 = var1 * 1      # var2: 1
```

```
var3 = var1 + var2  # var3: 2
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 re-computations. . .), hence the requirement of mechanisms for **glitch freedom**. Note that glitches are a consequence of inconsistent sequential 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 the previous listing, `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 particular, 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.

2.1.4 Functional Reactive Programming

Functional Reactive Programming (FRP) is a subset of both *reactive programming* and *functional programming*, retaining the advantages of the former, 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

[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 of this pattern.

term. Pure functions produce *no observable side effects* when applied and are **referentially transparent**, meaning that different applications of a function to the same inputs always produce the same outputs. To attain referential transparence, functions should avoid referencing *shared mutable data*, so that their behavior is kept constant since their definition and cannot have side effects (Listing 2.2).

```
def rt() = 10




// Application
val x = rt()            // 10
val y1 = x + x          // 10 + 10 = 20
val y2 = rt() + rt()    // 10 + 10 = 20
```

```
var g = 0     // mutable shared data
def ro() =
  g = g + 1   // observable side-effect
  10 * g

// Application
val x = ro()            // 10
val y1 = x + x          // 10 + 10 = 20
val y2 = ro() + ro()    // 20 + 30 = 50
```

Listing 2.2: An example of referentially transparent (*left*) and referentially opaque (*right*) functions. Note how referential transparence allows replacing any value with a call to the function that produced it.

In reactive programming, compositionality also requires glitch freedom, as observable glitches may invalidate the behavior expressed by a function. Indeed, functions may express different behaviors due to inconsistent handling of simultaneous events by the underlying evaluation model.

Compositionality is essential for dealing with complex software, addressing its complexity by combining simpler components that are easier to reason about. Moreover, it deals with scalable software, tackling its 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, namely Sodium, ScaFi, and FRASP, 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],

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

which is not glitch-free) [BJ16].

In Sodium, behaviors are modeled as **cells**, denoted by `Cell[V]`, which indicates a time-varying value of type `V`, while event streams are modeled as simply **streams**, denoted by `Stream[E]`, which indicates 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 2.3), creating and combining `Cells` and `Streams`. These include:

- `never`: create a new `Stream` that will never emit events.
- `map`: given a `Stream` s in input, create an output `Stream` s', whose events are the events of s, transformed with a given `mapping` function. An analogous operation is provided for `Cells`.
- `filter`: given a `Stream` s in input, create an output `Stream` s', whose events are the events of s, discarding those which do not satisfy a given `predicate`.
- `merge`: given two `Streams` s_1 and s_2 in input, create an output `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 in input, create an output `Stream` s', whose events are the events of s combined with the most recent value of c using a given `combine` function.
- `constant`: create an output `Cell` c, holding a given `value` forever.
- `hold`: given a `Stream` s in input, create an output `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 in input, 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 `Cell`s c_1 and c_2 in input, create an output `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 2.3: An abstract view on the Sodium primitives for constructing static acyclic computational graphs. Some primitives can be derived as a combination of the others (e.g., `constant` and `snapshot`).

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 `Stream`s or other `Cell`s). In particular, two switching operators are implemented in Sodium (Listing 2.4): `switchS` builds a dynamic computational graph from a `Cell` of `Stream`s c_s, creating an output `Stream` s', whose events are the events of the most recent `Stream` held by c_s; `switchC` works similarly for `Cell` of `Cell`s.

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

Listing 2.4: 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]): Cell[V]
```

Listing 2.5: 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 2.6), which are excluded from the core primitives, since their incorrect usage may break some properties of Sodium. A broker between a 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 pushing 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 computational graph). Conversely, the `listen` primitive implements FRP to non-FRP interactions, allowing the registration of 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 base 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 streamSink[E]: StreamSink[E]
def send[E](s: StreamSink[E], event: E): Unit
def listen[E](s: Stream[E], callback: E => Unit): Listener
def cellSink[V](initial: V): CellSink[E]
def send[V](c: CellSink[V], update: V): Unit
def listen[V](c: Cell[V], callback: V => Unit): Listener
```

Listing 2.6: 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 higher-order primitives to facilitate the construction of computational graphs (e.g., `accum`, `collect`, `sequence`, `gate`...). While these operators will not 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 helpful 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 concurrently 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 unnecessary CPU time consumption. 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 growing complexity of scalable 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 DSL for aggregate specifications and a platform for the simulation and execution of such specifications [CAV].

In ScaFi, the core concepts of field calculus are modeled by a **trait** like the one reported in Listing 2.7 [VBD+19], whose methods represent the constructs of field calculus.

```scala
trait FieldCalculus:
  // neighbors calculus
  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 2.7: 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 a 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 2.5).

Despite being equivalent, the semantics of ScaFi differs 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 leveraging 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` operator, which handles selective partitioning; evolution over time with `rep` follows the same semantics as in field calculus. This variant of field calculus assumes the name of **neighbors calculus** [ACDV23].

Additionally, ScaFi provides contextual operators that handle interactions with the underlying platform, namely `mid`, which computes the field of the device identifiers; `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).

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

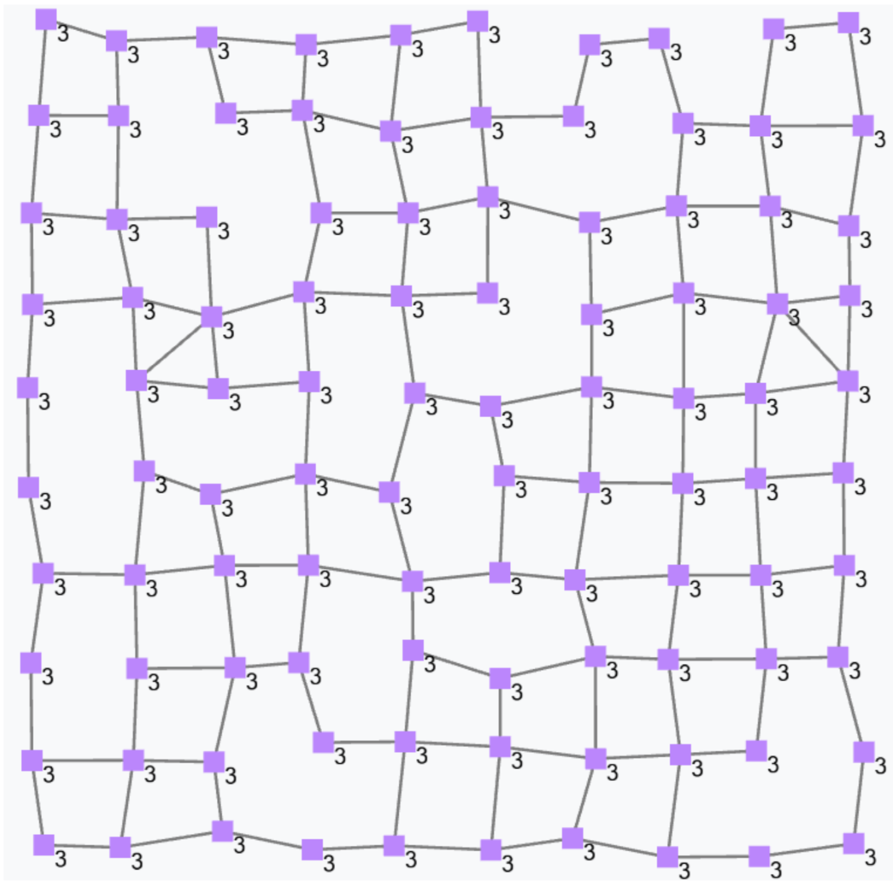


Figure 2.5: A graph representing an aggregate of devices (*nodes*) and their neighboring relations (*edges*). In particular, it represents the computational field yielded by the expression "1 + 2" (image from the site [CAV]).

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 2.8) and some derived operators (Listing 2.9).

```
trait AggregateComputing:
  self: FieldCalculus =>
  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
```

Listing 2.8: The core constructs of aggregate computing, represented as a mixin for field calculus, abstracting the actual organization within ScaFi.

```
trait ScafiLanguage:
  self: FieldCalculus with AggregateComputing:
  def branch[E](cond: => Boolean)(th: => E)(el: => E): E
```

```scala
  def mux[E](cond: => Boolean)(th: => E)(el: => E): E
  def share[E](exp: => E)(evolve: (E, () => E) => E): E
```

Listing 2.9: The core constructs of the ScaFi language, represented as a trait, abstracting over 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 a **round-based** execution model (Figure 2.6), in which a **round** is the computation required for an individual device to produce its next output based on the aggregate specification. Rounds are executed *asynchronously*, with timing determined by the scheduler of the platform. A round consists of the following three steps in order:

1. **Sense**: the device updates its current **context** (i.e., all known information from its perspective), by retrieving its *state* (i.e., its previous output), the information perceived through its *sensors* from the local environment and the messages transmitted by 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 subtrees 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 exports can be derived from the output of the device by searching in the abstract syntax tree for operations involving communication (e.g., subtrees 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]).

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

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

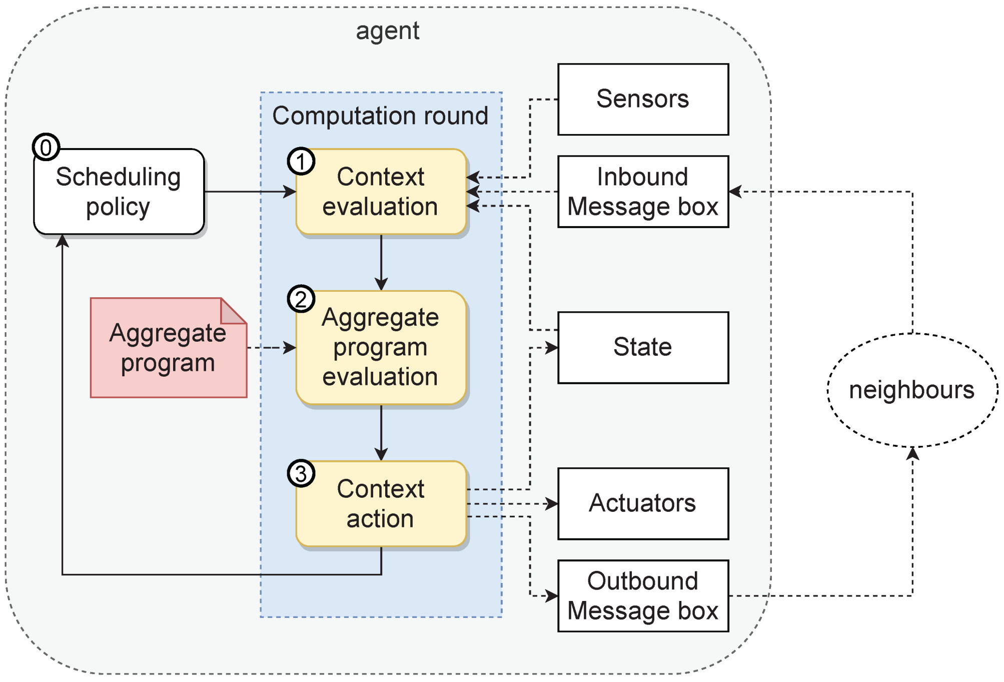


Figure 2.6: The round-based execution model of ScaFi (image from the paper [CAV21]). The diagram includes logical components (*solid rectangular boxes*), activities (*solid rounded boxes*), devices (*dashed rounded boxes*), flow of control (*solid arrows*), and flow of data (*dashed arrows*).

2.2.3 FRASP

FRASP (Functional Reactive Approach to Self-organization Programming)[7] is a novel open-source aggregate computing DSL 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:

- *Event-driven computation*: in a device, computation is driven by relevant

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

changes in its perception of the environment (e.g., sensors and neighbor data). As a result, 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, complete re-computations of an aggregate specification are avoided when possible.

- *Minimal communication*: a device only transmits its exports upon relevant changes, avoiding further message exchanges after the aggregate has reached a stable configuration. As a consequence, 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` keep track of their position inside the FRASP specification, building the abstract syntax tree used for alignment.

The syntax of FRASP faithfully resembles the syntax of field calculus, while also sharing common constructs with ScaFi (Listing 2.10). 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, `map` for unary operators and `lift` for binary operators (*explicit 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 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]
  def constant[V](value: V): Flow[V]
  def map[A, B](a: Flow[A])(operator: A => B): Flow[B]
  def lift[A, B, C](a: Flow[A], b: Flow[B])(operator: (A, B) => C): Flow[C]

  // 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 2.10: 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 2.7 and 2.8). On an abstract level, the simulator operates in two phases:

- **Configuration**: accept a FRASP specification, which describes the structure 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, based on the environment, and build the computational graph of the aggregate, based on the FRASP specification. In doing so, the simulator establishes the dependency chains from the percepts of each device (i.e., 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 may continue indefinitely even after reaching a stable state, since it is always possible for an event to happen in the future, however, no propagation of change implies no consumption of computational resources (i.e., the aggregate keeps waiting for an event to occur).

[8]An input node is 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.

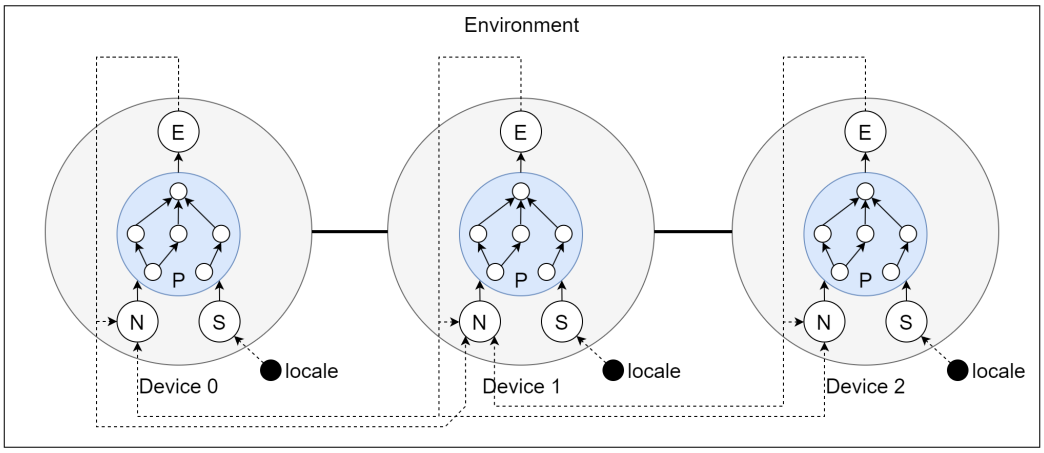


Figure 2.7: The reactive execution model of FRASP. In the diagram, three devices (*gray circles*) with neighboring relations (*solid lines*) are configured with an aggregate specification (*blue circles*, also denoted P). For each device, the input of the specification is neighboring (N) and local environmental data from sensors (S); the output is the export transmitted to neighbors (E). In the computational graph, there are internal (*solid arrows*) and external (*dashed arrows*) dependencies.

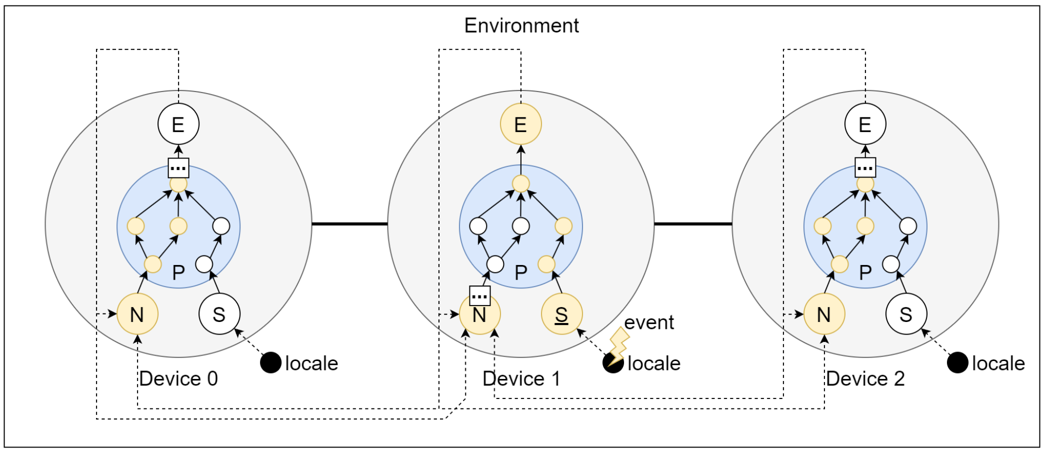


Figure 2.8: An example of propagation of change in the execution model of FRASP. The local environment of device 1 changed, causing changes to all its dependents. The three dots indicate that the change continues to propagate following the graph dependencies. Note how the propagation of change would carry on indefinitely in *any* cyclic graph without proper measures (e.g., calming pattern).

Since this project contributes to the implementation of FRASP, the following provides a brief overview of its architecture (Figure 2.9). Internally, FRASP is organized in the following three layered modules:

- `frp`: provide extensions and abstractions over the concrete FRP engine on which the framework depends.
- `core`: provide the model and implementation of the FRASP specification, as illustrated previously in Listing 2.10.
- `simulation`: provide a basic simulator for running aggregate specifications over a network of devices.

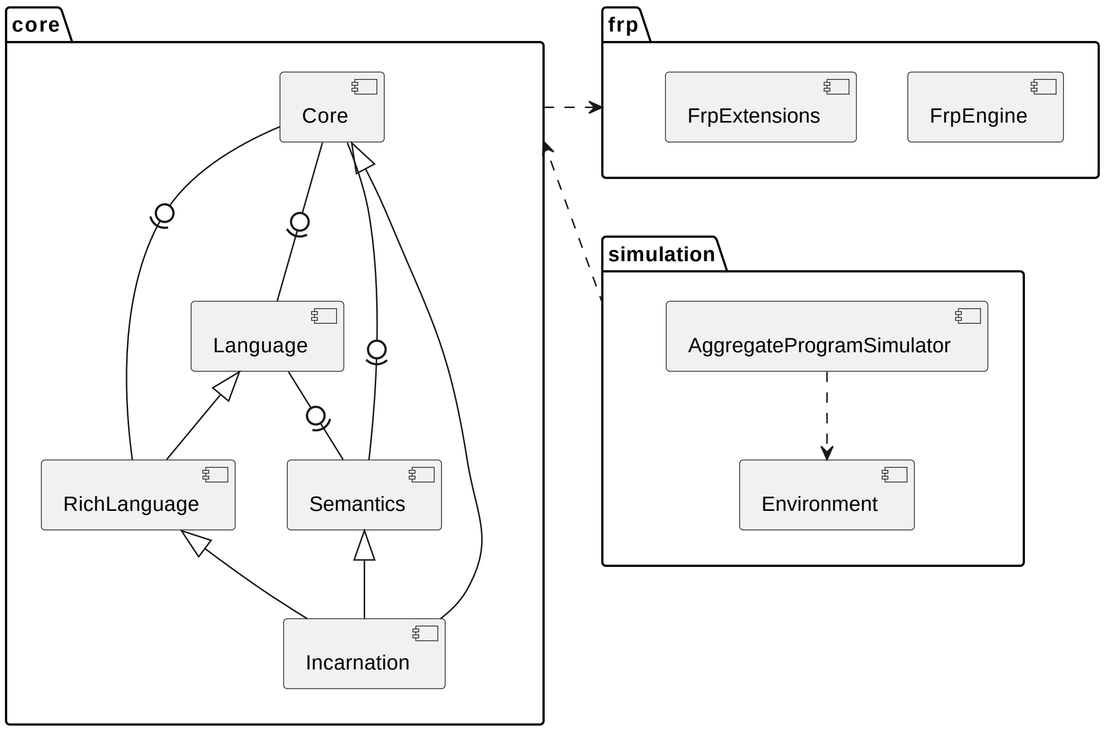


Figure 2.9: The architecture of FRASP (image from the paper [CDA+23]).

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

This chapter performs an analysis of the objectives and requirements of this project, outlining the strategy to achieve them. First, Section 3.1 introduces the objective of the project, which is the implementation of functional tests for FRASP. Then, Section 3.2 explains the process of testing aggregate specifications, while Section 3.3 describes the specific strategy adopted for testing aggregate specifications, mostly based on convergence properties. Finally, Section 3.4 concerns the verification of properties through simulations and analyzes the attributes of an ideal simulator.

3.1 Objectives

As anticipated in Section 1.1 of the introduction, FRASP is currently in research and requires a consolidated test suite supporting the research process, setting expectations on the behavior of the framework and assessing the correctness of its current state, so that software regressions may be avoided during its development. Additionally, the test suite could discover unexpected implications of the reactive model adopted by FRASP, as there are no guarantees on the equivalence between a reactive or round-based execution of the same specification.

At the current state, the test suite of the FRASP language contains only a few tests and samples verifying the reactivity and the generation of the device exports for each language construct individually (**semantic tests**), while functional tests concerning the execution of aggregate specifications (**aggregate tests**) are missing. In other words, there are no tests verifying that an aggregate evolves following the user's specification.

The goal of this project is to implement the missing aggregate tests, assessing the correctness of several specifications, executed against different network configurations and environments. ScaFi will be used as a reference for establishing the expectations on FRASP specifications, so that FRASP may be empirically proved functionally equivalent to ScaFi, while retaining its reactive benefits.

3.2 Aggregate Testing

An aggregate test should verify that an aggregate behaves as expected with respect to a given specification. However, while its purpose may be clear at an abstract level, a more detailed analysis is required to determine its concrete implications (e.g., what does it mean for an aggregate to behave as expected?).

Leveraging the network operational semantics of field calculus [VAB+18], the evolution of an aggregate can be described by a transition system in which each transition is $N_t \xrightarrow{act} N_{t+1}$ with the following notation:

$N_t ::= (\Psi_t, E_t)$	: the state of the aggregate at time t
$E_t ::= (\tau_t, \Sigma_t)$	: the state of the environment at time t
Ψ_t	: the output of all the devices at time t
τ_t	: the topology of the network at time t
Σ_t	: the percepts of the sensors at time t
$act ::= \delta_k$ or env	: a change in the aggregate
δ_k	: a change due to the k^{th} device transmitting its export
env	: a change in the environment

In a *static* environment E_0, transitions can be reduced to the form $\Psi_t \xrightarrow{\delta_k} \Psi_{t+1}$, i.e., the transition system is uniquely described by an initial aggregate state and the sequence of all the device exports.

Once the evolution of an aggregate is expressed as a transition system, formal verification techniques for transition systems may be applied to aggregates as well. In particular, one can express properties on aggregates using *propositional logic*, for static attributes, or even *temporal logics*, for dynamic or branching attributes. Then, properties may be verified through formal techniques such as *model checking* or *simulation*.

Properties are used to formally define the expectations for the evolution of an aggregate, including the output of the devices in the network, their percepts, the topology of the network, environmental changes and device communication. Expectations may involve one, some or all of the devices in the network, therefore properties can be:

- *Global*: a property of the whole aggregate (e.g., `mid` should evaluate to the identifiers of all the devices in the network).
- *Regional*: a property of a selected group of devices in an aggregate (e.g., `branch(isRed){obstacle}{somethingElse}` should evaluate to `obstacle` for all red devices in the network).
- *Individual*: a property of a single device in an aggregate (e.g., the 0^{th} device should always be a source of potential).

3.3 Aggregate Convergence Testing

The first step in consolidating the test suite is to implement several aggregate **unit tests**, considering a FRASP construct as the *software unit*, and **integration tests**, involving FRASP specifications (i.e., combinations of FRASP constructs). Best practices [Osh13] want unit tests to be:

- *Simple*: easy to implement. Indeed, inserting complex logic in a test requires such logic also to be tested, in order to ensure that the errors found by the test are not caused by faults in its logic. Moreover, simple tests can be easily understood, facilitating the detection of the cause of failure.
- *Isolated*: independent of other unit tests (i.e., concerning a single software unit). Dependencies between unit tests make it more difficult to detect the cause of failure.
- *Reproducible*: always yielding the same results under the same initial conditions (i.e., *determinism*). Non-determinism may cause a test to succeed under breaking changes or to fail even with no changes at all.
- *Finite*: yielding a result in a limited amount of time, ideally short for supporting frequent repeatability.
- *Automated*: executed each time a relevant (preferably small) increment of software is completed.

By definition, integration tests cannot be isolated, however the other properties should be preserved to the best of one's possibilities.

One of the challenges with aggregate tests is that the evolution of an aggregate is naturally non-deterministic, due to the unpredictability of communication in distributed systems. As a consequence, tests should be carefully designed to reason about some deterministic higher-level behavior exhibited by the underlying non-deterministic evolution of the aggregate (in literature, **don't care non-determinism**), so that they can be reproducible without forcing a deterministic evolution of the aggregate, which would be unrealistic and reduce the importance of the tests.

In this sense, the primary strategy employed in this project is **aggregate convergence tests**, which are based on the convergence of an aggregate towards an expected stable state. Convergence is a property that can be expressed in *linear temporal logic* as $\Diamond\Box P$ ("*sometimes P will hold forever*"). In particular, it may be interesting to evaluate the property $\Diamond\Box\Psi_{expected}$, understanding if the outputs of an aggregate will eventually reach and hold the expectation $\Psi_{expected}$. However, convergence can only be verified for self-stabilizing specifications (e.g., non-oscillating), assuming a finite number of changes in the environment.

3.4 Simulation

Since a simulator is already provided within FRASP, simulation will be used as a formal verification method for properties in aggregate tests. However, the current simulator is basic and lacks some properties required for adequate testing (referring to the previous Section 3.3). In particular, an adequate simulator for aggregate tests should provide the following properties:

- *Observability*: during a simulation, it should be possible for external entities to reconstruct the state of the simulation from its outputs. For aggregate tests, a simulation should expose at least the state of the aggregate and the progress of the simulation. Additionally, it would be useful to have access to individual, regional and global views of the aggregate. At the moment, the simulator does not expose any outputs to other entities, instead, it only shows the outputs of individual devices to the user through a console.

  This property is required for automated aggregate tests.

- *Controllability*: it should be possible to control a simulation, leading it to a stable state when its execution does not converge in a finite amount of time. For aggregate tests, a simulation should at least have the capability to be halted. At the moment, a simulation starts as the simulator is created and continues indefinitely, forever reacting to the next event.

  This property is required for finite aggregate tests.

- *Fairness*: in aggregate tests, it should always be possible for every device to compute an export in the future. At the moment, the simulator relies on the scheduling of the underlying runtime to achieve fairness.

This property is required to support self-stabilization.

- *Efficiency*: it should be optimized to minimize execution time, possibly leveraging parallelism. At the moment, the simulator supports concurrent execution, but it suffers from critical races (and other deeper problems discussed later in Section 4.4).

  This property is required to reduce the computational costs of testing and support frequent repeatability.

- *Reproducibility*: multiple simulations should yield similar results under similar initial configurations, implying the ability to execute a simulation under similar conditions multiple times (*repeatability*) or under different conditions (*replicability*).

  Obviously, this property is required for reproducible tests.

Since the current simulator does not provide all the aforementioned properties, an extension of the simulator is necessary. Most importantly, observability and controllability should be provided for testing. However, in doing so, one should be mindful of preserving the reactive execution model of FRASP as is. Indeed, a problem with testing through simulation is that the results of the tests may be influenced by the implementation of the simulator.

Chapter 4

Design

This chapter presents the abstract solution designed for achieving the objectives of this project. First, Section 4.1 introduces an extension of the base reactive execution model of FRASP with increased observability and controllability. Then, Section 4.2 provides a solution to the halting problem of FRASP simulations. Afterwards, Section 4.3 explains the process of performing aggregate convergence tests leveraging the new simulation models. Finally, Section 4.4 describes the parallelism constraints on concurrent simulations.

4.1 Simulation

The proposed approach for improving the observability and controllability of the current simulator is to provide an interface on top of the base reactive execution model of FRASP (Figure 4.1). The design of this interface abstracts from the underlying FRP framework, specifically Sodium, and from the approach adopted for managing the propagation of change in the computational graph (e.g., concrete implementations may support concurrency).

Regarding observability, in static environments, the evolution of an aggregate is uniquely described by its initial state and the sequence of device exports (as discussed in Section 3.2). As a consequence, complete observability of the evolution of an aggregate can be achieved by simply collecting all the exports in the order they were produced, whereas the initial state of the aggregate can be inferred from the specification and environment provided during the configuration of the reactive execution model (recall the configuration phase from Section 2.2.3). In practice, the simulation interface exposes a new reactive variable, called `exports`, derived by *merging* individual exports from each device. The firings of `exports` can also be filtered, mapped and accumulated to provide individual, regional or global views on the evolution of the aggregate.

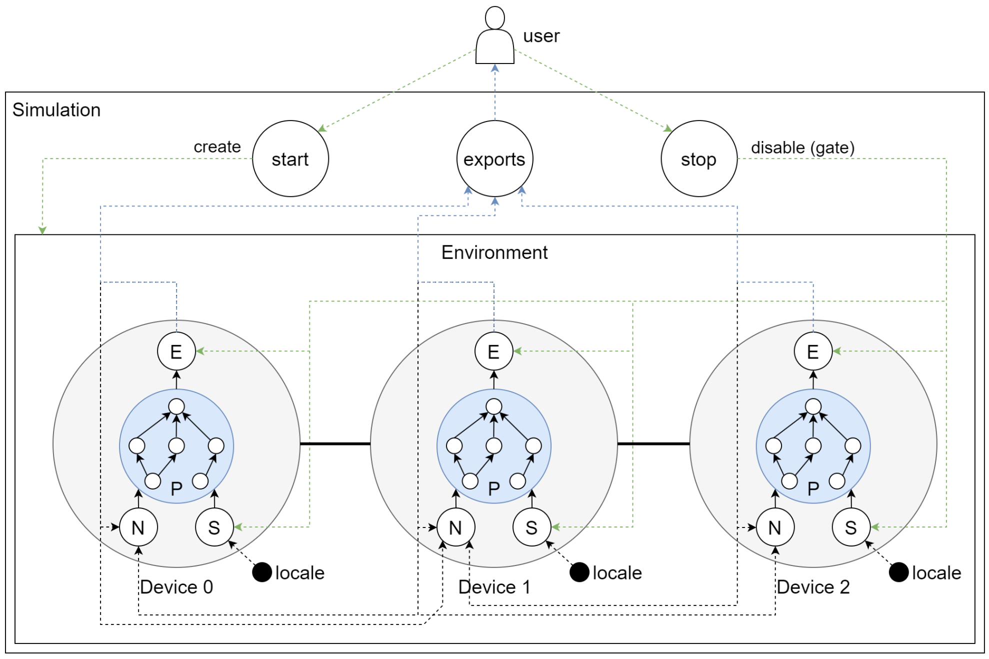


Figure 4.1: The interface of a simulation for observing and controlling the underlying reactive execution model of FRASP. The user can now observe the state of a simulation (`exports`), start it (`start`) or stop it (`stop`). To support these new functionalities, new dependencies have been added for observability (*blue dashed arrows*) and controllability (*green dashed arrows*).

Observability extends to dynamic environments by leveraging device sensors (using the `sensor` and `nbrSensor` constructs) to reactively produce exports containing changes in the environment.

Concerning controllability, the simulation interface offers a `start` functionality, delaying the creation of the computational graph of the base reactive execution model until requested by the user (instead of building the graph when the simulation is created). This delay allows the user to declare their own dependencies on the `exports` of the simulation before its execution. As a consequence, forward referencing is required for `exports` to declare its dependencies on the individual exports of the devices.

To stop a simulation, it is possible to filter any future export when requested by the user, freezing all views on the simulation (i.e., `exports`) and blocking any communication within the aggregate. However, non-observable computation could still be triggered following a change in the environment, even after the simulation is stopped. Therefore, to optimize the simulation, any future percept of the device sensors should also be filtered upon termination. Alternatively, the computational graph should be disposed of, if possible.

An important aspect to consider for the observability of a simulation is *simulation time*, which measures the progress of a simulation. In event-driven simulations, time is quantified as the number of events fired since the beginning of the simulation (i.e., the number of firings of the variable `exports`). Still, this representation has some implications, such as time not advancing if no events are fired, affecting controllability. Indeed, one cannot rely on the simulation time to stop the simulation without prior knowledge about the evolution of the aggregate. For example, halting a simulation after a specific number of events is unreliable, because it cannot be assumed that the simulation will ever fire that many events, so the condition may never be satisfied. However, a similar policy may be required to ensure termination in aggregate tests, when the simulations never reach a stable state, perhaps due to the nature of the specification or an unknown flaw in its implementation.

Abstracting from the specific use case of halting the simulation, the problem is that neither the user nor the simulation have an understanding of the progress of the aggregate's evolution. On the one hand, the user lacks information about the number of events that will be produced in the simulation, which depends on the concrete implementations of the simulation and specifications. On the other hand, the simulation lacks information about possible changes of the environment, that may be triggered not only by the device actuators, but also by external entities, such as the user. As a consequence, none of the parties can evaluate when the evolution of the aggregate should be considered concluded. This issue is referred to as the **halting problem** in the following sections.

4.2 Step Simulation

In order to address the halting problem, the previous simulation interface should be refined to attain increased observability and controllability, providing the user with more information about the state of the simulation. One possibility is to model the concept of **step simulation**, which would allow the user to execute a simulation step-by-step, receiving feedback after every step (Figure 4.2).

The concept behind a step simulation involves keeping track of the exports of the devices, deferring their propagation until requested by the user. This strategy achieves greater observability since the simulation knows precisely the number of pending exports, allowing the user to be notified when there is none. Controllability is also increased, as the user can decide exactly when the simulation should continue. Ultimately, the halting problem is solved if the user has complete control over the environment. Indeed, in this scenario, the user can reasonably assume that the evolution of the aggregate has concluded (and the simulation should be

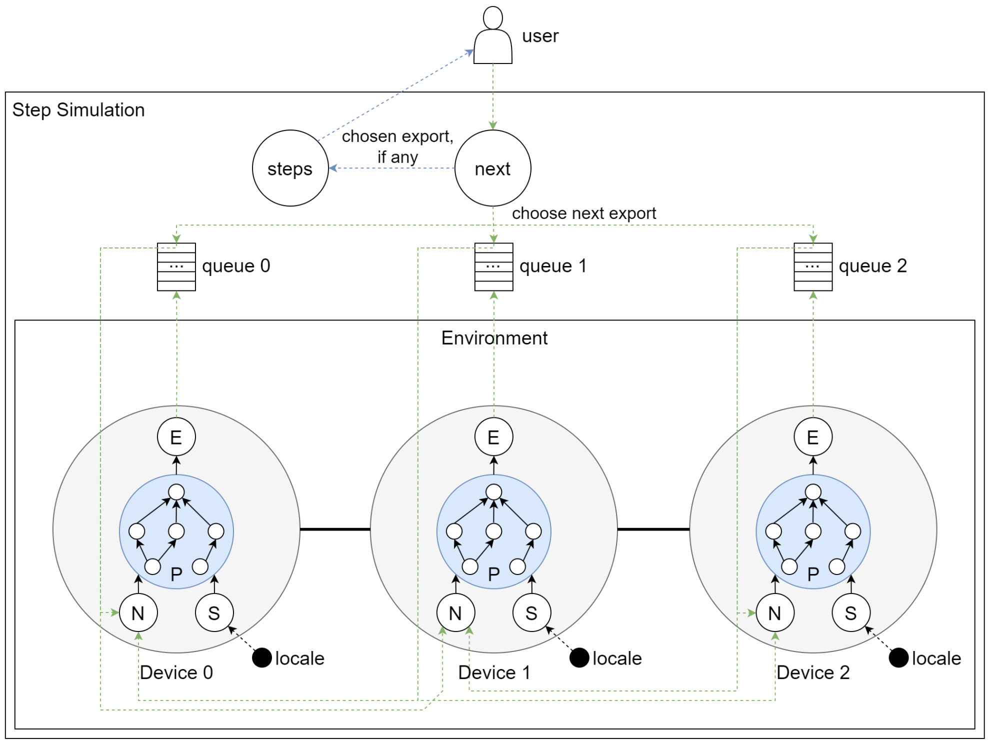


Figure 4.2: The interface of a step simulation. Each device owns a queue of exports that need to be transmitted. In fact, when an export is produced, it is inserted in the queue of the corresponding device, instead of being broadcast directly to its neighbors. The interface exposes new functionalities for selecting, transmitting (`next`) and observing (`steps`) the next export from the queues, increasing observability (the user can be notified if there are no more exports to be transmitted) and controllability (the user can decide exactly when the simulation should continue). The other functionalities of simulations are supported, but hidden for clarity.

stopped) when there are no more exports to propagate and no further intention to change the environment.

In detail, the simulation maintains a queue of exports for each device, so that the device exports are pushed into the queue of the corresponding device when produced. The user can request the execution of the next **step** (i.e., the propagation of the next export), then an export is extracted from one of the device queues and transmitted to the neighbors. For each request, the user is notified of the extracted export or the lack of pending exports through the reactive variable `steps`. Note how the system behaves exactly like the base reactive model of FRASP if a request is sent as soon as a new export is produced.

An added benefit of this approach is the ability to manage the scheduling of device exports. When a user request is received, the simulation takes charge of selecting the next export to transmit. To this end, various scheduling policies can be implemented. For instance, the next export can be extracted from one of the non-empty queues chosen randomly, ensuring non-determinism in the aggregate's evolution. Alternatively, a round-robin policy can be employed to select the next export, ensuring fairness in the simulation.

Concurrency is also supported as events from different export queues may be propagated at the same time. However, synchronization is required to guarantee a consistent view of the simulation from the perspective of the user.

4.3 Convergence Simulation

The simulation interface can be extended once more to provide direct support for aggregate convergence tests, exposing an operation for evaluating the limit of an aggregate evolution with respect to time, that is a stable state for self-stabilizing specifications. To support such operation, during the configuration phase, a simulation should accept a **halt policy**, which is a condition that stops the simulation when satisfied. Moreover, such halt policy should be designed so that the simulation is stopped when the aggregate reaches a stable state.

For example, for a step simulation, a suitable halt policy would be to stop the simulation when there are no more exports that can be extracted from the device queues. Instead, for a general reactive simulation, a suitable halt policy would be to stop the simulation after a certain period of inactivity (i.e., time elapsed since the last emitted event). However, real-world time introduces non-determinism in the results of the simulations, rendering the policy not suitable for testing.

4.4 Concurrent Simulation

Concurrency in reactive simulations can be achieved by delegating the propagation of change to some workers (e.g., a thread pool). In particular, in Sodium, concurrency can be achieved by removing a dependency between a consumer and a producer in a computational graph, then listening to the events of the producer and delegating to a worker the propagation of each event towards the consumer. However, this approach is limited to concurrency and cannot achieve parallelism, due to Sodium's transactional system.

As discussed in Section 2.2.1, Sodium's transactions are executed one at a time to guarantee glitch freedom, trading off parallelism to ensure consistency. Later, it was discovered that transactions are executed sequentially even among independent computational graphs. Therefore, unless the computation of a device is detached from the computational graph (i.e., executed outside the FRP engine), concurrent simulations cannot be executed in parallel, in fact concurrent events are still processed sequentially.

Moreover, the deployment of the reactive execution model in real distributed systems is still unclear, possibly hinting towards the exploration of distributed reactive programming solutions. Further research could discover the effects of Sodium's consistency in the evolution of aggregate of devices and evaluating the possibility of achieving the same level of consistency in large-scale distributed systems, such as CASs.

Chapter 5

Implementation

This chapter describes the concrete solution implemented for this project, building upon the design presented in the previous chapter. First, it introduces the implementations of several simulators, including a general simulator (Section 5.1), a step-by-step simulator (Section 5.2), a concurrent simulator (Section 5.3), and a convergence simulator (Section 5.4). Then, Sections 5.5 and 5.6 detail two extensions of the Sodium library, integral to the implementation of the simulators and the test suite. Finally, Section 5.7 discusses a solution for supporting dynamic environments in FRASP.

5.1 Simulator

The implementation of a simulator is depicted by the class diagram in Figure 5.1, which is based on the design described in Section 4.1.

A `Simulator` can be used to create several `Simulations`, each one requiring a `Flow`, that is a FRASP specification, and a `Configuration`, which includes the `Environment` where the aggregate is situated. To define the concepts of `Flow` and `Environment`, a `Simulator` relies on a specific `SimulationIncarnation`, which is an instance of the aggregate computing DSL in FRASP, tailored for simulation.

The implementation of the `Simulator` follows the *cake pattern*, in which dependencies are defined inside external mixin components and can be imported by extending the desired components. In particular, `Simulator` depends on the `SimulationConfigurationComponent`, which defines the concept of `SimulationConfiguration`, and the `SimulationComponent`, which defines the concept of `Simulation`. In the following sections, specializations of `Simulator` defines additional concepts using other components, adhering to the same naming convention.

As per design, a `Simulation` provides observability by means of the methods `exported`, which supplies a `Stream` of all the device exports transmitted within

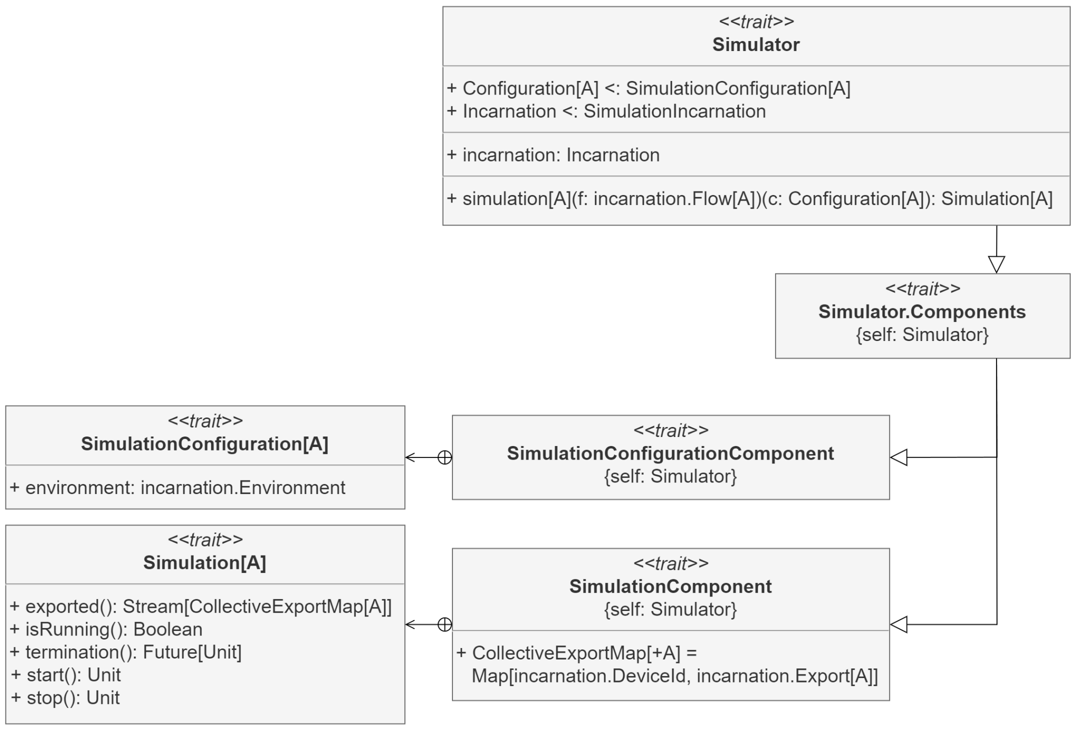

<<trait>>
Simulator
+ Configuration[A] <: SimulationConfiguration[A]
+ Incarnation <: SimulationIncarnation
+ incarnation: Incarnation
+ simulation[A](f: incarnation.Flow[A])(c: Configuration[A]): Simulation[A]
<<trait>>
Simulator.Components
{self: Simulator}
<<trait>>
SimulationConfiguration[A]
+ environment: incarnation.Environment
<<trait>>
SimulationConfigurationComponent
{self: Simulator}
<<trait>>
Simulation[A]
+ exported(): Stream[CollectiveExportMap[A]]
+ isRunning(): Boolean
+ termination(): Future[Unit]
+ start(): Unit
+ stop(): Unit
<<trait>>
SimulationComponent
{self: Simulator}
+ CollectiveExportMap[+A] =
Map[incarnation.DeviceId, incarnation.Export[A]]

Figure 5.1: A UML class diagram of the simulator and its components.

the aggregate, `exportedBy`, which supplies a `Stream` of all the exports of a single device (*an individual view*), and `exportedByAll`, which supplies a `Stream` of all the device exports accumulated during the simulation (*a global view*, in which each event is a `CollectiveExportMap`, that is a map from the devices to their latest export). Similar methods are provided to observe only the result of the computations, that is the root of the device exports.

Concerning controllability, a `Simulation` exposes one method `start` to begin its execution, running the underlying `startBehavior` of the concrete type of `Simulation`, and another method `stop` to halt it, running the underlying `stopBehavior` likewise. Moreover, a method `isRunning` can be used to know if the simulation has already started but has not stopped yet, while another method `termination` allows reacting to the end of the simulation.

5.2 Step Simulator

The implementation of a step simulator is represented by the class diagram in Figure 5.2, which is based on the design described in Section 4.2.

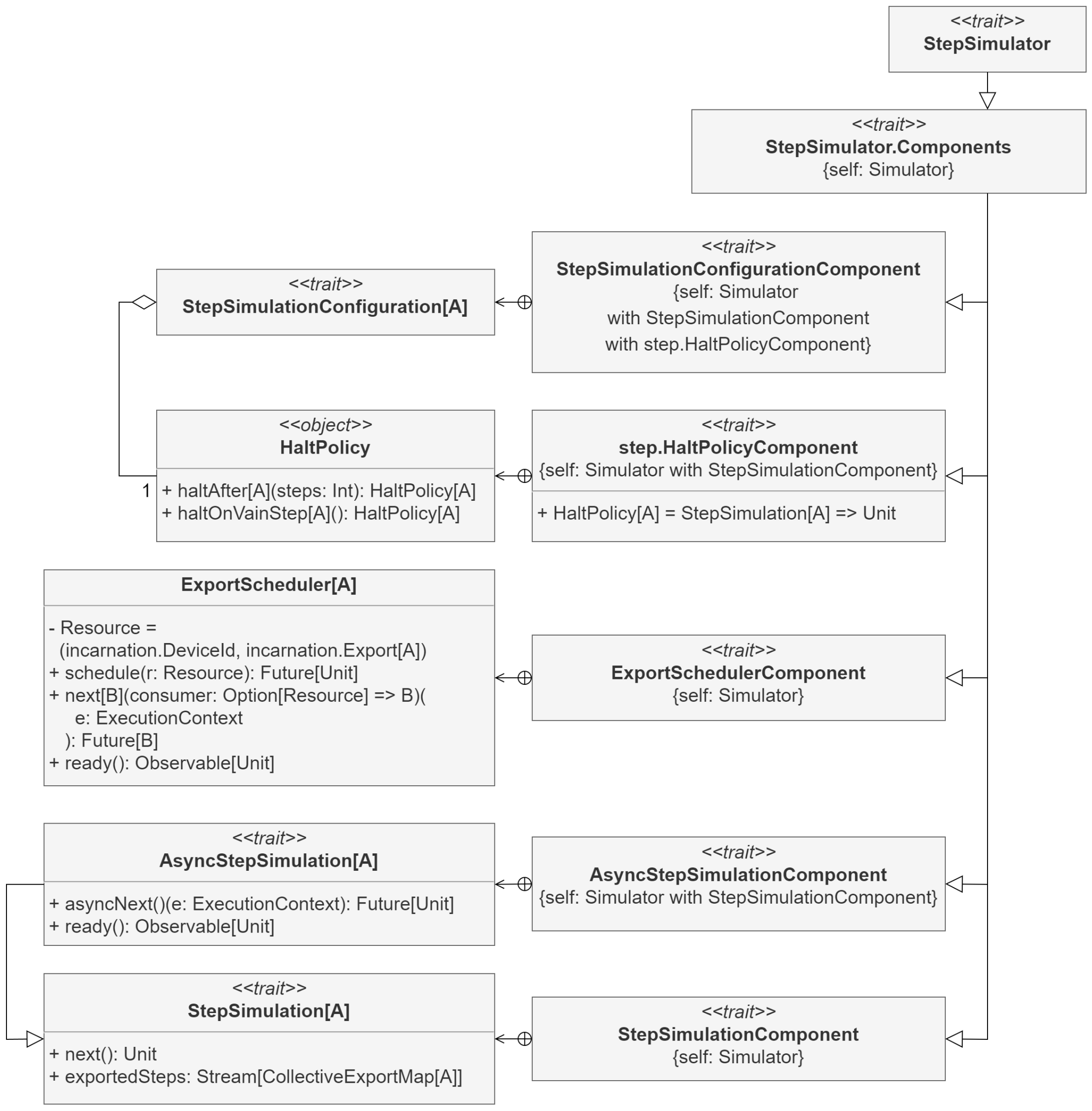

<<trait>>
StepSimulator
<<trait>>
StepSimulator.Components
{self: Simulator}
<<trait>>
StepSimulationConfigurationComponent
{self: Simulator
with StepSimulationComponent
with step.HaltPolicyComponent}
<<trait>>
StepSimulationConfiguration[A]
<<object>>
HaltPolicy
1
+ haltAfter[A](steps: Int): HaltPolicy[A]
+ haltOnVainStep[A](): HaltPolicy[A]
<<trait>>
step.HaltPolicyComponent
{self: Simulator with StepSimulationComponent}
+ HaltPolicy[A] = StepSimulation[A] => Unit
ExportScheduler[A]
- Resource =
(incarnation.DeviceId, incarnation.Export[A])
+ schedule(r: Resource): Future[Unit]
+ next[B](consumer: Option[Resource] => B)(
e: ExecutionContext
): Future[B]
+ ready(): Observable[Unit]
<<trait>>
ExportSchedulerComponent
{self: Simulator}
<<trait>>
AsyncStepSimulation[A]
+ asyncNext()(e: ExecutionContext): Future[Unit]
+ ready(): Observable[Unit]
<<trait>>
AsyncStepSimulationComponent
{self: Simulator with StepSimulationComponent}
<<trait>>
StepSimulation[A]
+ next(): Unit
+ exportedSteps: Stream[CollectiveExportMap[A]]
<<trait>>
StepSimulationComponent
{self: Simulator}

Figure 5.2: A UML class diagram of the step simulator and its components.

A `StepSimulator` is a `Simulator` that creates `StepSimulations`. As per design, a `StepSimulation` provides enhanced observability and controllability by means of the methods `next`, which executes a step of the simulation by transmitting the next device export to the neighbors and the user, and `exportedSteps`, which supplies a `Stream` of the device exports transmitted at each step.

A thread-safe variant of `StepSimulation` is the `AsyncStepSimulation`, which provides a new method `asyncNext`, executing the next step of the simulation in a given `ExecutionContext`, and a new property `ready`, allowing registering callbacks to execute each time a new export is available for transmission. Actually, only an implementation of `AsyncStepSimulation` has been developed at the moment, meaning that every `StepSimulation` is an `AsyncStepSimulation` under the hood. In the future, a specific implementation of `StepSimulation` may be developed to be optimized for single-threaded execution.

Specifically, the implemented `AsyncStepSimulation` involves keeping track of the device exports through per-device queues, deferring their transmission to the neighbors until requested by the user, that is when the methods `next` or `asyncNext` are called, which delegate the propagation of change to the user or to an `ExecutionContext` respectively. The device queues are managed by an `ExportScheduler`: each time an export is produced, the method `schedule` of the scheduler is called, queuing the device export for later consumption; each time the next step of the simulation is requested by the user, the method `next` of the scheduler is called, dequeuing and transmitting the next export both to the neighbors and the user. The transmission to the user happens by pushing the exports into the `exportedSteps` stream. If all the device queues are empty, an empty event is pushed instead.

The `ExportScheduler` decides the order of transmission of the device exports, preserving the order of computation for each device (i.e., the export of a device cannot be transmitted before its previous export). In particular, the scheduling policy adopted by the current implementation is a best-effort round-robin, in which each device is given the same chance to transmit its exports as long as they have some to transmit, guaranteeing fairness during the simulation.

The `SimulationConfiguration` of a `StepSimulation` is modeled by the class `StepSimulationConfiguration`. In addition to the `Environment`, the configuration includes an `HaltPolicy`, which contains the logic for determining the end of the simulation. Some built-in `HaltPolicy`s are already defined in the corresponding simulator component, namely: `never`, which never halts the simulation (the user may still stop it at any time); `haltWhen`, which halts the simulation when a given predicate holds for the state of the aggregate; `haltAfter`, which halts the simulation after a given number of steps; finally, `haltOnVainStep`, which halts the simulation when a step is executed, but all the export queues are empty. Additionally, `HaltPolicy`s may be merged by means of the `combine` operator to consider multiple conditions of termination, halting the simulation when any of them is satisfied.

The `StepSimulationConfiguration` is interpreted by the `StepSimulation` when the `start` method is called, creating the devices of the aggregate, building its computational graph, scheduling the first exports and setting up the `HaltPolicy`.

Example. A practical application of the `StepSimulator` is demonstrated in the following program (Listing 5.1).

```scala
// Creation
object Incarnation extends SimulationIncarnation:
  override type Environment = environment.Environment
object Simulator extends StepSimulator, WithIncarnation(Incarnation):
  override type Configuration[A] = StepSimulationConfiguration[A]
import Simulator.incarnation.{*, given}

// Configuration
val configuration = Simulator.StepSimulationConfiguration[DeviceId](
  environment = environment.Environment.euclideanGrid(cols = 3, rows = 3),
  haltPolicy = Simulator.HaltPolicy.haltOnVainStep,
  logger = Logger.NoOperation,
)
val simulation = Simulator.simulation[DeviceId](mid)(using configuration)

// Preparation
simulation.exportedSteps.listen(step => println(step))
simulation.termination.onComplete(_ => println("END"))

// Execution
simulation.start()
while(simulation.isRunning){ simulation.next() }
```

Listing 5.1: An application of `StepSimulator`. The simulator is used to display the device exports on the standard output.

5.3 Concurrent Simulator

The implementation of a concurrent simulator is described by the class diagram in Figure 5.3.

A `ConcurrentSimulator` is simply a `Simulator` whose `Simulations` are executed concurrently. A basic implementation of a concurrent `Simulation` can be developed by leveraging an underlying `AsyncStepSimulation`. In detail, the transmission of the device exports to the neighbors and the user is delegated to an `ExecutionContext`, which schedules the computation on a thread pool. Transmission is scheduled as soon as an export is generated, that is whenever the underlying simulation is `ready`. As a consequence, from the perspective of the user, the concurrent simulation behaves exactly the same as the base reactive model of FRASP, without the increased observability of the step simulation, which would have required further research to be developed due to the inherent challenges of concurrency. Such development has been postponed since the concurrency of the simulation did not translate to parallelism due to Sodium's transactions, as better discussed in Section 4.4 of the design.

The `SimulationConfiguration` of a concurrent `Simulation` is modeled by the class `ConcurrentSimulationConfiguration`. In addition to the `Environment` and

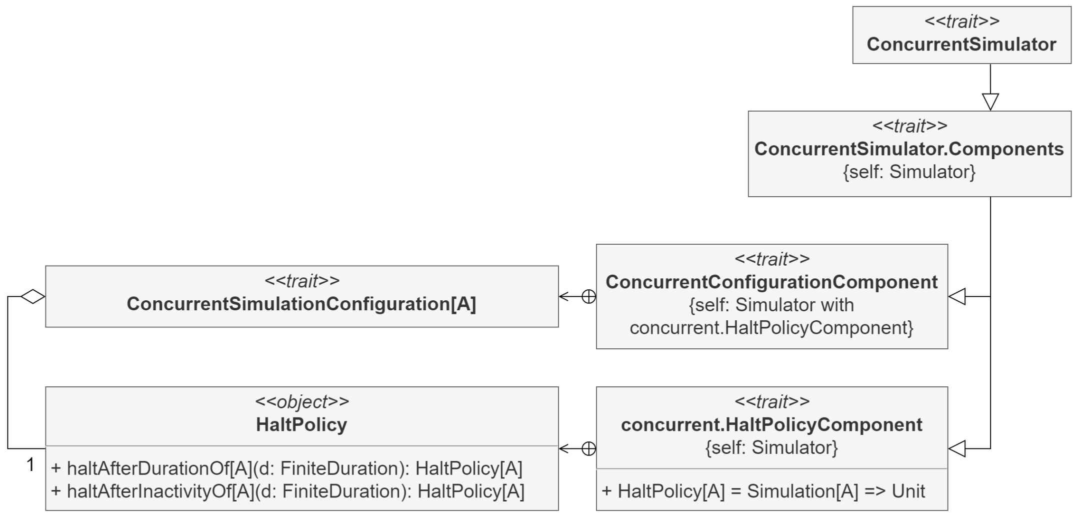

<<trait>>
ConcurrentSimulator
<<trait>>
ConcurrentSimulator.Components
{self: Simulator}
<<trait>>
ConcurrentSimulationConfiguration[A]
<<trait>>
ConcurrentConfigurationComponent
{self: Simulator with concurrent.HaltPolicyComponent}
<<object>>
HaltPolicy
+ haltAfterDurationOf[A](d: FiniteDuration): HaltPolicy[A]
+ haltAfterInactivityOf[A](d: FiniteDuration): HaltPolicy[A]
1
<<trait>>
concurrent.HaltPolicyComponent
{self: Simulator}
+ HaltPolicy[A] = Simulation[A] => Unit

Figure 5.3: A UML class diagram of the concurrent simulator and its components.

`HaltPolicy`, the configuration includes the `ExecutionContext` where the simulation will be executed. Some built-in `HaltPolicys` are already defined in the corresponding simulator component, including `never`, `haltWhen` and two others, namely `haltAfterDurationOf`, which halts the simulation after a certain period of time has elapsed since its start, and `haltAfterInactivityOf`, which halts the simulation after a certain period of time has elapsed since its latest event.

Example. A practical application of the `ConcurrentSimulator` is demonstrated in the following program (Listing 5.2).

```scala
// Creation
object Incarnation extends SimulationIncarnation:
  override type Environment = environment.Environment
object Simulator extends ConcurrentSimulator, WithIncarnation(Incarnation):
  override type Configuration[A] = ConcurrentSimulationConfiguration[A]
import Simulator.incarnation.{*, given}

// Configuration
val executor = Executors.newFixedThreadPool(nThreads = 10)
val configuration = Simulator.ConcurrentSimulationConfiguration[DeviceId](
  environment = environment.Environment.euclideanGrid(cols = 3, rows = 3),
  haltPolicy = Simulator.HaltPolicy.haltAfterInactivityOf(5.seconds),
  executor = ExecutionContext.fromExecutor(executor),
  logger = Logger.NoOperation,
)
val simulation = Simulator.simulation[DeviceId](mid)(using configuration)

// Preparation
simulation.exported.listen(exported => println(exported))
simulation.termination.onComplete(_ => executor.shutdown())

// Execution
```

```
23 simulation.start()
```

Listing 5.2: An application of `ConcurrentSimulator`. The program is very similar to Listing 5.1, however, note that the concurrent simulator accepts a different configuration (line 10). Additionally, the exports are generated continually by the provided `ExecutionContext` after the simulation is started (after line 23, there is no `next` method to call).

5.4 Convergence Simulator

The implementation of a convergence simulator is illustrated by the class diagram in Figure 5.4.

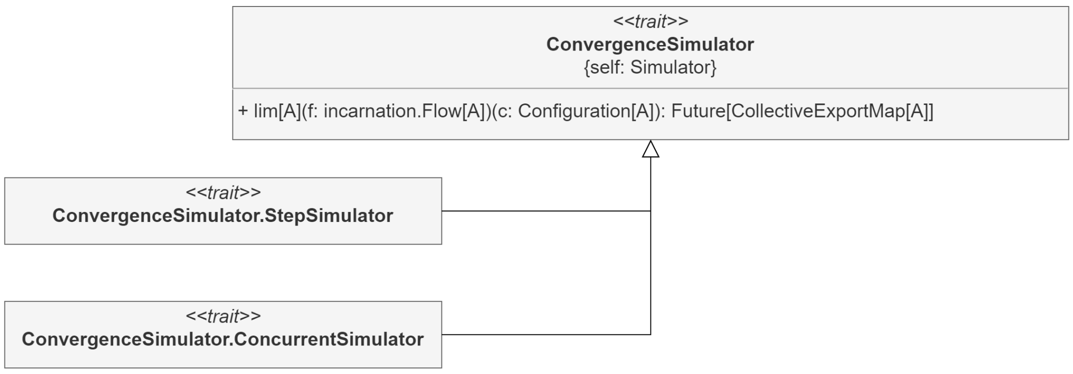

<<trait>>
ConvergenceSimulator
{self: Simulator}
+ lim[A](f: incarnation.Flow[A])(c: Configuration[A]): Future[CollectiveExportMap[A]]
<<trait>>
ConvergenceSimulator.StepSimulator
<<trait>>
ConvergenceSimulator.ConcurrentSimulator

Figure 5.4: A UML class diagram of the convergence simulator and its concrete types.

A `ConvergenceSimulator` is a `Simulator` with the additional ability of evaluating the stable state of an aggregate in self-stabilizing specifications, namely the method `computeLimit` (or its shorthand `lim`). The idea behind `computeLimit` is as simple as executing a simulation until its termination, returning the last export transmitted by each device in the network, in the form of a `CollectiveExportMap`. Observation of the environment is possible by leveraging the device sensors within the specification, attaching their percepts to the device exports. However, to ensure that the result of `computeLimit` is a stable state of the aggregate, the specification should be self-stabilizing and the simulation should be halted after the aggregate has stabilized. Additionally, if the aggregate has multiple possible non-deterministic stable states, `computeLimit` can only evaluate one of them. Indeed, it may be useful to consider only the root of the device exports to reduce the number of stable states of the aggregate, possibly to a single deterministic stable state.

Given a self-stabilizing specification, the result of `computeLimit` is determined by the `HaltPolicy` in the configuration specified by the user. Below follows an analysis of the suitable `HaltPolicy`s and their effects, assuming the user has complete control over the environment:

- `haltWhen(_ ==` N_s`)`: calling N_s the *known* stable state of the aggregate, the simulation can be halted when the aggregate is in state N_s. This policy works for all simulations, however, it can only be used for evaluating the **reachability** of the state N_s ("sometimes N_s holds"), which is a looser property compared to convergence ("sometimes N_s holds forever").
- `haltAfterDurationOf(T)`: if a stable state exists, after an infinite amount of time the aggregate will have certainly stabilized. In general, the longer a simulation is executed (i.e., T), the higher the chances of the aggregate having stabilized. This policy works for all simulations, however, relying on real-world time introduces non-deterministic results.
- `haltAfterInactivityOf(T)`: inactivity in the simulation can be interpreted as a symptom of stability in the aggregate. In general, the longer the simulation is inactive (i.e., T), the higher the chances of the aggregate having stabilized. This policy works the same as `haltAfterDurationOf`, albeit possibly being more flexible (e.g., the same value of T may apply to a larger variety of simulations).
- `haltAfter(N)`: similar to `haltAfterDurationOf`, but it relies on the steps of a simulation in order to track the simulation time. This policy offers deterministic results for deterministic simulations, however, it can only be used for `StepSimulation`s.
- `haltOnVainStep`: this policy guarantees to halt the simulation when the aggregate has stabilized, however, it can only be used for `StepSimulation`s.

At the moment, two concrete implementations of `ConvergenceSimulator` are available, based on the `StepSimulator` and `ConcurrentSimulator`.

Example. A practical application of the `ConvergenceSimulator` is demonstrated in the following program (Listing 5.3).

```
// Creation
object Incarnation extends SimulationIncarnation:
  override type Environment = environment.Environment
object Simulator
  extends ConvergenceSimulator.StepSimulator, WithIncarnation(Incarnation):
  override type Configuration[A] = StepSimulationConfiguration[A]
import Simulator.incarnation.{*, given}
import Simulator.lim
```

```
 9
10  // Configuration
11  given Simulator.StepSimulationConfiguration[DeviceId] =
12    Simulator.StepSimulationConfiguration(
13      environment = environment.Environment.euclideanGrid(cols = 2, rows = 2),
14      haltPolicy = Simulator.HaltPolicy.haltOnVainStep,
15      logger = Logger.NoOperation,
16    )
17
18  // Execution
19  def count(from: Int, to: Int): Flow[Int] =
20    loop(from)(_.map(c => math.min(c + 1, to)))
21  lim(count(from = 0, to = 10))  // Map(0 -> 10, 1 -> 10, 2 -> 10, 3 -> 10)
22  lim(count(from = 5, to = 10))  // Map(0 -> 10, 1 -> 10, 2 -> 10, 3 -> 10)
23  lim(count(from = 5, to = 15))  // Map(0 -> 15, 1 -> 15, 2 -> 15, 3 -> 15)
```

Listing 5.3: An application of `ConvergenceSimulator`. The program evaluates the stable states for three similar specifications (lines 21-23), in which each device counts all the integer numbers in a given range. For simplicity, the stable states only show the root of the devices exports.

5.5 Stream Extension

In support of the simulators and the test suite, the set of operations on `Streams` provided by Sodium has been extended with new operators for the manipulation, analysis, and monitoring of `Streams`. In designing these operators, care was taken to preserve compositionality, implementing them as pure functions, and fluency, integrating them into the existing `Stream` class by means of Scala's *extension methods*, also because inheritance of the base types of `Sodium` is not allowed.

The collection of implemented extension methods is provided by the `StreamExtension` object, which defines the operators discussed in the following sections.

5.5.1 Persistence Operators

The following operators can be used to deal with state persistence in `Streams`:

- `collect`: evolve an initial state `init` as the input `Stream` s generates new events and return an output `Stream` s', whose events are a combination of the events fired by s with the current state at the moment of firing. The given `accumulator` function defines both the evolution of the state and the events of s' depending on the firings of s.

  The `collect` operator is an adaptation for Scala of the homonymous operator provided by `Sodium` in Java.

- `fold`: a simplification of the `collect` operator, in which the events of the output `Stream` s' are a snapshot of the current state, taken at each firing of the input `Stream` s. The given `accumulator` function defines the evolution of the state depending on the firings of s, and the events of s' as a consequence.

The `fold` operator is an implementation of the *folding* operation provided by Scala for all `Iterables`. However, the events of the input `Stream` are folded lazily as they are fired.

An example of their application is the evolution of the global view of an aggregate, such as the method `exportedByAll`, obtained by accumulating the individual device exports generated during a simulation.

5.5.2 Temporal Operators

The following operators can be used to perform time-sensitive analysis on `Streams`:

- `zipWithIndex`: when applied to an input `Stream` s, return an output `Stream` s', whose events are the same events of s paired with the discrete time when they were fired. Discrete time is modeled as the number of firings preceding an event in s'.

- `zipWithTime`: when applied to a `Stream` s, return an output `Stream` s', whose events are the same events of s paired with the continuous time when they were fired. Continuous time is defined by a given `Clock`, which defaults to the number of nanoseconds elapsed since the creation of s'. Abstracting over real-world time allows the user to provide their own implementation of `Clock` to achieve complete control on the timeline of the `Stream`, which can be useful to avoid non-determinism during tests.

- `zipWithDelay`: when applied to an input `Stream` s, return an output `Stream` s', whose events are the same events of s paired with the continuous time elapsed since the previous event. Similarly to `zipWithTime`, continuous time is defined by a given `Clock`.

An example of their application is the implementation of time-sensitive `Halt-Policys`, such as the `haltAfter` policy, introduced in Section 5.2.

5.5.3 Derivation Operators

The following operators can be used to perform trend and behavior analysis on `Streams`:

- `ngrams`: when applied to an input `Stream` s, return an output `Stream` s', whose events are all the possible groups of consecutive events fired by s with cardinality n. Note that s' does not fire any event until s has emitted at least n events, which may never happen. In such case, any information about the events of s is lost in s'.

- `ngramsOption`: as `ngrams`, but information loss is prevented by producing incomplete groups in s' until s has generated at least n events. An incomplete group contains all the consecutive events fired by s and as many placeholder values needed to reach cardinality n. In particular, `Option.None` is used as a placeholder value.

A future application of these operators could be the verification of more sophisticated temporal properties against the evolution of aggregates, checking the behavior of the aggregate during a fixed time-window (e.g., evaluating if the sum of the outputs of all the devices is always the sum at the previous step plus one).

5.5.4 Monitoring Operators

The following operators can be used to monitor the events generated by `Stream`s:

- `cold`: when applied to an input `Stream` s, return an output `Stream` s', whose events are sequences containing all the firings of s after the creation of s'. Since s may fire events indefinitely, the length of the sequences can be limited to a given amount of `memory`, which corresponds to the number of events of s kept in memory by the operator. When the memory is full, the oldest events are replaced with the newest ones as they are fired.

  The name of this operator comes from the notion of **cold observable**s, as described in Section 6.2.1 of the book [BJ16]. In Sodium, all `Stream`s are inherently **hot observable**s, meaning that any dependent will react only to the events that are fired after its dependency has been declared, ignoring all the events that were fired before. The `cold` operator creates a `Stream` that acts *almost* as a cold observable variant of the original `Stream`, by letting the dependents react also to the events that were fired before the declaration of their dependencies. However, dependents will be notified of all the events of the original `Stream` only after its next firing, which may never happen. To solve this problem, the `Stream` can be transformed into a `Cell` by means of the `hold` operator, obtaining an actual cold observable. In fact, `Cell`s propagate their latest state as soon as a dependent is declared.

- `monitor`: when applied to an input `Stream` s, return a `StreamMonitor` wrapping s. A `StreamMonitor` relies on the `cold` operator to monitor the wrapped

`Stream`, exposing a sequence of all its events, accessible at any time by means of the method `eventLog`. In other words, a `StreamMonitor` converts a `Stream` into an up-to-date list of its events. Similarly to the `cold` operator, the length of the sequence can be limited to reduce memory costs.

The purpose of these operators is to decouple the generation of the events of a `Stream` from the evaluation of their properties, which greatly simplifies testing. However, performing the evaluation during the generation of the events would be more efficient, as the program generating the events could be interrupted prematurely if the property was already proven before the program termination. Naturally, this optimization cannot be implemented by leveraging these operators, since the evaluation starts only after the program termination.

An example of their application is the implementation of most of the test suite and the `ConvergenceSimulator`, in which the global view of the aggregate is monitored to return its latest state when the simulation is halted.

5.5.5 Throttling Operators

The following operators can be used to control the throughput of `Streams`:

- `sync`: combine two input `Streams` s_1 and s_2, returning an output `Stream` s', whose events are the pairs of corresponding events in s_1 and s_2. More formally, the k^{th} event of s' is a pair containing the k^{th} event of s_1 and the k^{th} event of s_2. Since s_1 and s_2 may fire their k^{th} event at different times, the operator requires keeping in memory the latest unpaired events of both `Streams`. Similarly to the `cold` and `monitor` operators, the number of events stored for each `Stream` can be limited to a given `memory`.

  An implication of the `sync` operator is that the throughput of the output `Stream` is equal to the lowest throughput between the input `Streams`, meaning that the operator can be used to control the frequency at which a `Stream` emits its events. Additionally, by limiting the memory of the operator, some events of the `Stream` with the highest throughput may be discarded when the memory is full, preventing possible overloads of its dependents.

- `throttleWith`: a specialization of the `sync` operator with unitary `memory`, storing only the latest unpaired event of each input `Stream`.

- `throttle`: a specialization of the `throttleWith` operator, in which the output `Stream` s' fires the events of the first input `Stream` s_1, while the second input `Stream` s_2 acts only as a throttle for s_1.

A future application of these operators could be regulating the event production rate of reactive variables in general, including `Streams`, `Cells` and possibly `Flows`.

5.6 Finite Stream Extension

Another extension implemented for the Sodium library is the `FiniteStreamExtension`, which defines a new type of reactive variable derived from `Streams`, namely the `FiniteStream` type.

In Sodium, `Streams` may fire new events indefinitely, making them suitable for modelling any type of producer. However, in their generality, `Streams` do not capture directly the fact that some producers only emit a finite amount of events. For this purpose, `FiniteStreams` have been designed to fire a finite amount of events before notifying all the dependents of their termination.

Since extending `Streams` is not allowed in Sodium, `FiniteStreams` have been implemented as `Streams` of `FiniteEvents`, which can either be `Events`, wrapping a payload, or an `EOS (End Of Stream)`, marking the termination of the stream. This implementation allows leveraging the `Stream` operators also for `FiniteStreams`, however, it does not restrict producers from emitting additional events after the first `EOS`. To solve this problem, every event following the first `EOS` is automatically discarded.

The `FiniteStreamExtension` provides a set of operators for creating `FiniteStreams`, implemented in the form of Scala's extension methods, similarly to the `StreamExtension`. For better compositionality, these operators have been designed to transform `FiniteStreams` into other `FiniteStreams`. In fact, if the operators were transformations from `Streams` to `FiniteStreams`, they would wrap the firings of an input `Stream` inside the `FiniteEvents` of an output `FiniteStream`. However, they could also be applied to `FiniteStreams`, wrapping the `FiniteEvents` of an input `FiniteStream` inside the `FiniteEvents` of an output `FiniteStream`. As a consequence, any combination of operators would create a `Stream` of nested `FiniteEvents`, requiring recursive unnesting to access the actual payload of the events. For this reason, the only operator converting `Streams` into `FiniteStreams` is the entry point of the extension, namely the `finite` operator, which simply wraps any event of an input `Stream` inside an `Event` of an output `FiniteStream`.

The sole purpose of `finite` is to enable the application of the other operators of the extension, namely:

- `until`: when applied to an input `FiniteStream` s, return an output `FiniteStream` s', obtained by halting s at the first event whose payload satisfies a given `predicate`.

- `take`: when applied to an input `FiniteStream` s, return an output `FiniteStream` s', obtained by halting s after a given number of events.

- `interruptBy`: when applied to an input `FiniteStream` s, return an output `FiniteStream` s', obtained by halting s at the first event of a given `interruptor` stream.

- `takeBefore`: when applied to an input `FiniteStream` s, return an output `FiniteStream` s', obtained by halting s at the first event fired after a given `duration` has elapsed since the creation of s'.

- `interruptAfter`: when applied to an input `FiniteStream` s, return an output `FiniteStream` s', obtained by halting s after a given `duration` has elapsed since the creation of s'. The operator relies on a `Timer` to generate a notification after a set amount of time.

- `takeBeforeInactivityOf`: when applied to an input `FiniteStream` s, return an output `FiniteStream` s', obtained by halting s at the first event fired after a given `duration` has elapsed since its latest event.

- `interruptAfterInactivityOf`: when applied to an input `FiniteStream` s, return an output `FiniteStream` s', obtained by halting s after a given `duration` has elapsed since its latest event. The operator relies on a `Timer`, similarly to `interruptAfter`.

An example of application of the `FiniteStreamExtension` is the implementation of several `HaltPolicy`s for simulations, including `haltAfterDurationOf` and `haltAfterInactivityOf` for concurrent simulations.

5.7 Dynamic Environments

In support of the test suite, specifically for tests concerning sensors, an explicit model of dynamic environments has been implemented. In fact, FRASP provided only an explicit model of static environments, while changes in the environment were supported with ad-hoc mechanisms, involving direct modification of the `SimulationIncarnation` used to define the specifications.

A basic implementation of a dynamic environment is the `EnvironmentWithTags` (Figure 5.5), which is an `Environment` where devices can be linked to specific bits of information, called *tags*. In particular, the methods `tag` and `untag` allow attaching and detaching tags from a set of devices, while the method `withTag` can be used to retrieve the time-varying set of the devices linked to a specific tag.

With the introduction of dynamic environments, the `SimulationIncarnation` had to be updated to depend on an environment type, instead of an environment instance (Figure 5.6). Otherwise, the same `SimulationIncarnation` could not be used to define different specifications, as any program would be executed on

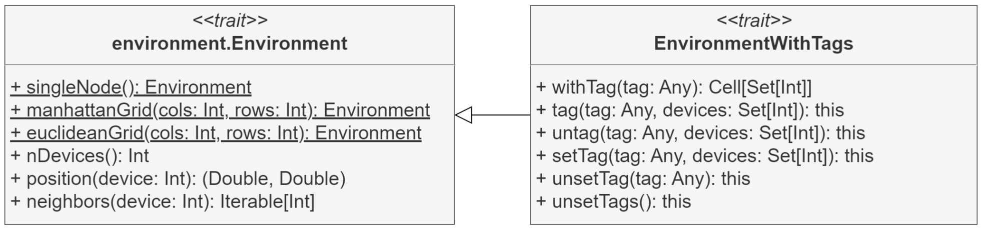

<<trait>>
environment.Environment
+ singleNode(): Environment
+ manhattanGrid(cols: Int, rows: Int): Environment
+ euclideanGrid(cols: Int, rows: Int): Environment
+ nDevices(): Int
+ position(device: Int): (Double, Double)
+ neighbors(device: Int): Iterable[Int]
<<trait>>
EnvironmentWithTags
+ withTag(tag: Any): Cell[Set[Int]]
+ tag(tag: Any, devices: Set[Int]): this
+ untag(tag: Any, devices: Set[Int]): this
+ setTag(tag: Any, devices: Set[Int]): this
+ unsetTag(tag: Any): this
+ unsetTags(): this

Figure 5.5: A UML class diagram of the environment types available for simulation.

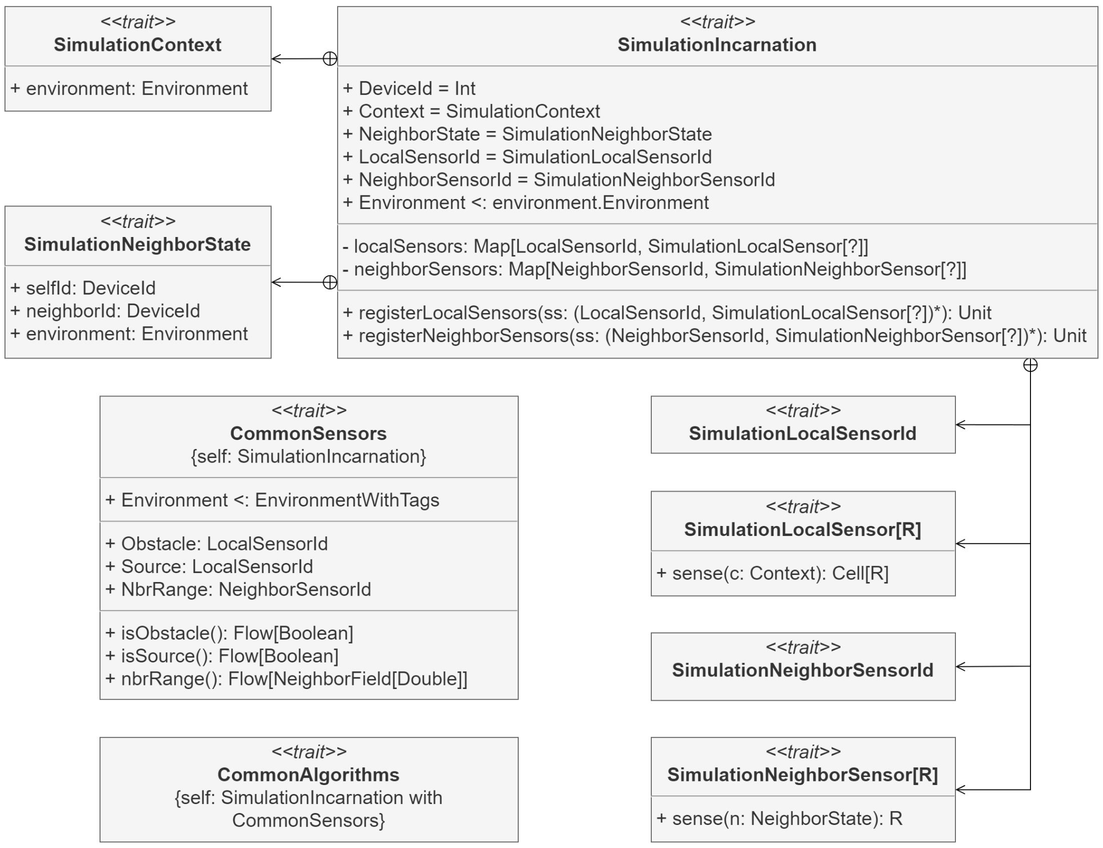

<<trait>>
SimulationContext
+ environment: Environment
<<trait>>
SimulationIncarnation
+ DeviceId = Int
+ Context = SimulationContext
+ NeighborState = SimulationNeighborState
+ LocalSensorId = SimulationLocalSensorId
+ NeighborSensorId = SimulationNeighborSensorId
+ Environment <: environment.Environment
- localSensors: Map[LocalSensorId, SimulationLocalSensor[?]]
- neighborSensors: Map[NeighborSensorId, SimulationNeighborSensor[?]]
+ registerLocalSensors(ss: (LocalSensorId, SimulationLocalSensor[?])*): Unit
+ registerNeighborSensors(ss: (NeighborSensorId, SimulationNeighborSensor[?])*): Unit
<<trait>>
SimulationNeighborState
+ selfId: DeviceId
+ neighborId: DeviceId
+ environment: Environment
<<trait>>
CommonSensors
{self: SimulationIncarnation}
+ Environment <: EnvironmentWithTags
+ Obstacle: LocalSensorId
+ Source: LocalSensorId
+ NbrRange: NeighborSensorId
+ isObstacle(): Flow[Boolean]
+ isSource(): Flow[Boolean]
+ nbrRange(): Flow[NeighborField[Double]]
<<trait>>
CommonAlgorithms
{self: SimulationIncarnation with CommonSensors}
<<trait>>
SimulationLocalSensorId
<<trait>>
SimulationLocalSensor[R]
+ sense(c: Context): Cell[R]
<<trait>>
SimulationNeighborSensorId
<<trait>>
SimulationNeighborSensor[R]
+ sense(n: NeighborState): R

Figure 5.6: A UML class diagram of the new `SimulationIncarnation`.

the same environment, already modified by previous programs, possibly causing unpredictable results.

Additionally, the `SimulationIncarnation` has been improved to support the registration of local and neighbor sensors, retrieving localized information from the environment, instead of relying on information injected directly into the `Simu-`

`lationIncarnation`. In detail, local sensors are modeled by the trait `SimulationLocalSensor`, which is a function producing the readings of a given device in the environment. Local sensors can be registered to the `SimulationIncarnation` under a specific identifier, typed `SimulationLocalSensorId`, by which they can be referenced using the `sensor` construct within an aggregate specification. Similarly, neighbor sensors are implemented by the traits `SimulationNeighborSensor` and `SimulationNeighborSensorId`.

To enhance modularization and isolation of concerns, the implementation of the sensors has been extracted from the `SimulationIncarnation` and delegated to specific traits (Figure 5.7). In particular, sensors are modeled by the trait `Sensor`, which declares the type of environment where the sensor can be employed, namely `SuitableEnvironment`. A `Sensor` can be either a `LocalSensor`, creating the corresponding `SimulationLocalSensor` for suitable `SimulationIncarnations`, or a `NeighborSensor`, creating the corresponding `SimulationNeighborSensor` for suitable `SimulationIncarnations`. A `SimulationIncarnation` is suitable for a `Sensor` if the `Environment` of the incarnation is a `SuitableEnvironment`.

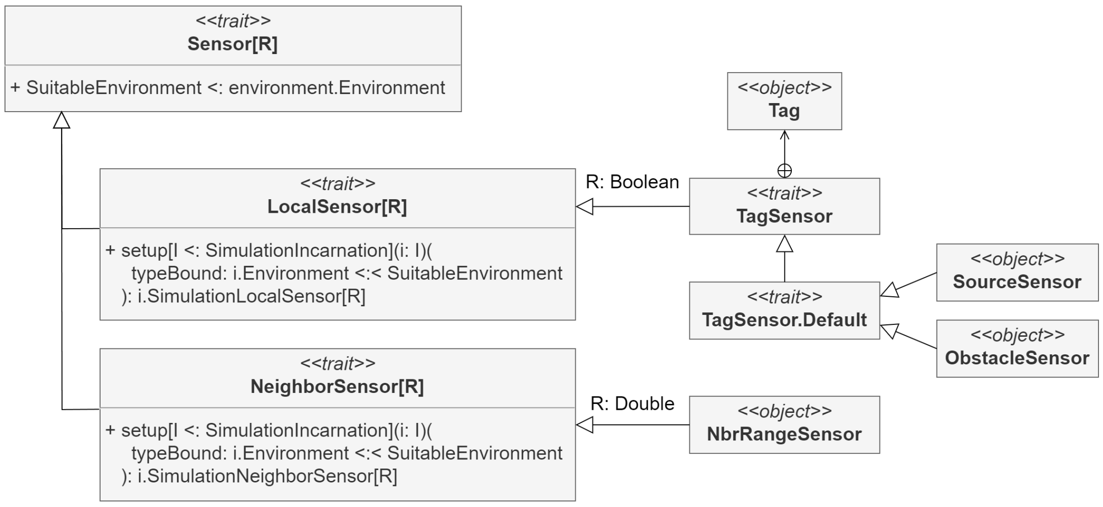

<<trait>>
Sensor[R]
+ SuitableEnvironment <: environment.Environment
<<object>>
Tag
R: Boolean
<<trait>>
LocalSensor[R]
+ setup[I <: SimulationIncarnation](i: I)(
typeBound: i.Environment <:< SuitableEnvironment
): i.SimulationLocalSensor[R]
<<trait>>
TagSensor
<<trait>>
TagSensor.Default
<<object>>
SourceSensor
<<object>>
ObstacleSensor
<<trait>>
NeighborSensor[R]
+ setup[I <: SimulationIncarnation](i: I)(
typeBound: i.Environment <:< SuitableEnvironment
): i.SimulationNeighborSensor[R]
R: Double
<<object>>
NbrRangeSensor

Figure 5.7: A UML class diagram of the sensor types available for simulation.

Some built-in sensors have already been implemented, namely `TagSensor`, which is a `LocalSensor` detecting if a specific tag is linked to a device (e.g., if the device is marked as an obstacle or a source), and `NbrRangeSensor`, which is a `NeighborSensor` measuring the distances from a device and all of its neighbors.

Finally, leveraging these concepts, two mixins for `SimulationIncarnations` have been implemented, namely the `CommonSensors` mixin, extending the DSL with a set of standard sensors, and the `CommonAlgorithms` mixin, extending the DSL with a set of gradient-based algorithms.

Chapter 6

Verification

This chapter describes the test suite developed for the verification of the FRASP language, delving into the implementation of aggregate convergence tests in Section 6.1 and analyzing the results of these tests in Section 6.2.

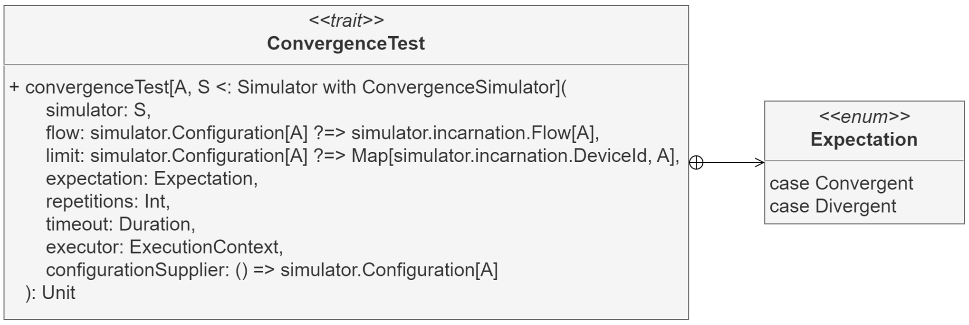
6.1 Unit and Integration Testing

The test suite developed for FRASP consists of around 250 tests, comprising unit and integration tests, a quarter of which concerns the verification of aggregate specifications, while the remaining tests verify the infrastructure built for the evaluation of properties on aggregates, including the simulators, the extensions for Sodium, and some utilities employed within the library. The tests have been implemented considering the best practices discussed in Section 3.3 of the analysis, leveraging the ScalaTest framework [Art].

As already anticipated in the analysis, the adopted strategy for verifying the correctness of the FRASP language is aggregate convergence tests, checking if the evolution of an aggregate converges towards an expected stable state, given an aggregate specification and a simulation configuration. Such tests are modeled by the mixin `ConvergenceTest` (Figure 6.1), which can be extended to provide the inheriting class with methods for creating tests based on convergence, namely:

- `convergenceTest`: using a provided `ConvergenceSimulator`, check if the evolution of an aggregate converges towards (or diverges from) an expected stable state, given an aggregate specification and a configuration for the simulation. The test can be repeated multiple times for statistical significance, since the evolution of the aggregate may be non-deterministic. Additionally, a timeout can be set to interrupt the test in case a simulation continues indefinitely against expectations (e.g., due to a flaw in the implementation of a specification).

- `convergentEquivalenceTest`: similar to `convergenceTest`, but it checks if different aggregate specifications converge to the same limit.



<<trait>>
ConvergenceTest
+ convergenceTest[A, S <: Simulator with ConvergenceSimulator](
simulator: S,
flow: simulator.Configuration[A] ?=> simulator.incarnation.Flow[A],
limit: simulator.Configuration[A] ?=> Map[simulator.incarnation.DeviceId, A],
expectation: Expectation,
repetitions: Int,
timeout: Duration,
executor: ExecutionContext,
configurationSupplier: () => simulator.Configuration[A]
): Unit
<<enum>>
Expectation
case Convergent
case Divergent

Figure 6.1: A UML class diagram of the convergent tests.

In detail, the test suite contains unit `ConvergenceTests`, concerning basic specifications with a single construct, and integration `ConvergenceTests`, concerning more complex specifications with multiple interacting constructs, including standard aggregate algorithms, such as gradient-based algorithms. The verified specifications have been collected in a mixin for `SimulationIncarnation`, called `FraspSamples`, granting access to their documentation and declarative names for referencing them.

Example. A practical application of a `ConvergenceTest` is demonstrated in the following program (Listing 6.1), which encapsulates all the concepts introduced so far. Note how the test is essentially a verbose set of initial conditions and an expectation, with no complex logic at all. Moreover, since the majority of tests share the same initial conditions, most of the complexity of the configuration can be eliminated via default parameters, promoting simplicity and standardization.

```scala
class GradientTest extends AnyFlatSpec with Matchers with ConvergenceTest:
  object Incarnation
    extends SimulationIncarnation with CommonSensors with CommonAlgorithms:
    override type Environment = EnvironmentWithTags
  object Simulator
    extends ConvergenceSimulator.StepSimulator, WithIncarnation(Incarnation):
    override type Configuration[A] = StepSimulationConfiguration[A]
  import Simulator.incarnation.{*, given}

  given configurationSupplier[A]: () => Simulator.Configuration[A] = () =>
    Simulator.StepSimulationConfiguration[A](
      environment = EnvironmentWithTags(euclideanGrid(cols = 5, rows = 5)).tag(
        tag = SourceTag,
        devices = Set(0)
      ),
      haltPolicy = Simulator.HaltPolicy.haltOnVainStep,
    )
```

```
18
19    "The gradient specification" should
20    "compute the field of distances from a source" in
21    convergenceTest(
22      simulator = Simulator,
23      flow = gradient(sources = sensor(Source)),
24      limit = Map(
25         0 -> 0.00,  1 -> 1.00,  2 -> 2.00,  3 -> 3.00,  4 -> 4.00,
26         5 -> 1.00,  6 -> 1.41,  7 -> 2.41,  8 -> 3.41,  9 -> 4.41,
27        10 -> 2.00, 11 -> 2.41, 12 -> 2.83, 13 -> 3.83, 14 -> 4.83,
28        15 -> 3.00, 16 -> 3.41, 17 -> 3.83, 18 -> 4.24, 19 -> 5.24,
29        20 -> 4.00, 21 -> 4.41, 22 -> 4.83, 23 -> 5.24, 24 -> 5.66,
30      ),
31      expectation = Expectation.Convergent,
32      repetitions = 10,
33      timeout = 60.seconds,
34    )
```

Listing 6.1: An application of `ConvergenceTest`. First, at lines 2-8, the test prepares the aggregate computing DSL, extended with standard sensors (such as the `Source` sensor) and algorithms (such as the `gradient`). Also, a `ConvergenceSimulator` is created, using the `StepSimulator` variant. Then, at lines 9-16, all simulations are configured to execute in an `EnvironmentWithTags`, specifically a grid-like network topology of 25 devices, where the 0^{th} device is tagged as a source. Additionally, a proper `HaltPolicy` for convergence is selected. Finally, at lines 18-33, a concrete convergence test is formulated: the test involves a gradient originating from all the devices tagged as sources (i.e., only the 0^{th} device); the evolution of the aggregate is expected to be convergent towards the given limit; and the test is iterated 10 times, failing automatically if convergence is not proven within 60 seconds.

6.2 Observations

The test suite yielded positive results, overall proving the FRASP language to be working as expected. However, the tests also confirmed a couple of issues, described in the following sections.

6.2.1 Referential Transparence

The aggregate computing DSL in FRASP enjoys referential transparence, granting compositionality for its constructs. As a consequence of referential transparence, the same behavior can be defined by replacing the values of a program with the constructs producing those values, like in the following example.

```
def collectNeighbors: Flow[Set[DeviceId]] =
  nbr(mid).map(_.values.toSet)

```

```
4 def branchAndCollectNeighbors(cond: Flow[Boolean]): Flow[Set[DeviceId]] =
5   branch(cond){ collectNeighbors }{ collectNeighbors }
6
7 def collectNeighborsAndBranch(cond: Flow[Boolean]): Flow[Set[DeviceId]] =
8   val neighbors = collectNeighbors
9   branch(cond){ neighbors }{ neighbors }
```

The above listing defines three specifications: `collectNeighbors`, for collecting the set of neighbors of all the devices (line 1); `branchAndCollectNeighbors`, for partitioning the network and then collecting the set of neighbors in each partition (line 4); finally, `collectNeighborsAndBranch`, for collecting the set of neighbors of all devices and then partitioning the network (line 7). Unexpectedly, `branchAndCollectNeighbors` and `collectNeighborsAndBranch` yield the same results, due to referential transparence (line 5 is semantically equivalent to lines 8-9). In summary, `collectNeighborsAndBranch` does not work as intended.

While referential transparence is not at all a negative property, FRASP currently has no mechanisms for referencing the results of other specifications within the local scopes of its constructs. In the example, the forks of a `branch` construct cannot access the set of neighbors collected before partitioning. This was a known issue[1], confirmed by the test suite of FRASP.

6.2.2 Inconsistency of Loop

The test suite found some specifications based on the `loop` construct producing inconsistent results. The inconsistencies manifested as an infinite execution for non-trivial self-stabilizing specifications, expected to stabilize in a finite amount of time. After further analysis, it was discovered that the inconsistent specifications had in common an evolution over time dependent both on the previous value P and at least one external source of change S (e.g., the `nbr` or `sensor` constructs).

The cause of the problem is attributed to the fact that the previous and current simulators do not propagate exports immediately, but their transmission is scheduled for later in the future. However, the implementation of the `loop` construct relies on self-communication for evaluating the previous value P. As a consequence, there is a delay between the generation of the new value of `loop` and the corresponding update of P.

During the delay, other dependencies of the `loop` construct, such as S, may trigger a propagation of change, generating a new inconsistent value of `loop`, based on the stale value of P and the new value of S. The problem is aggravated by the fact that the inconsistent value is also scheduled for transmission, including self-communication. As a result, any such inconsistent value starts an inconsistent evolution over time, concurrently with the others. This phenomenon can quickly

[1]https://github.com/cric96/distributed-frp/issues/1

build up, causing the simulation to never cease firing new events.

Analyzing this issue was especially difficult, since most affected specifications in the test suite, including the gradient, achieve a robust convergence even with the inconsistent values, probably due to the type of operations involved in the specification (e.g., accumulating information by evaluating the minimum value).

An example of highly susceptible specification is the following.

```scala
case class Round[T](time: Int, result: Option[T])
object Round { def zero[T]: Round[T] = Round(time = 0, result = None) }

def zipWithRound[T](flow: Flow[T]): Flow[Round[T]] =
  loop(Round.zero[T]) { rounds =>
    lift(rounds, flow) {
      case (Round(time, current), next) if !current.contains(next) =>
        Round(time + 1, Some(next))
      case (currentRound, _) =>
        currentRound
    }
  }
```

The specification in the example, namely `zipWithRound`, simply creates a `Flow` whose exports are the exports of the input `Flow`, bound to the time when they are transmitted (*round*), ignoring repeated consecutive exports. However, if the input `Flow` has an external dependency (e.g., a `nbr` construct), the output `Flow` will likely emit inconsistent values. This specification nicely demonstrates the issue, as inconsistent values are manifested as repeated or even oscillating rounds in the exports.

A solution to this issue could be to implement self-communication as an instantaneous transmission, however, a concrete implementation is yet to be developed.

Chapter 7

Conclusions

This thesis started with the goal of providing proper verification for the functionalities of FRASP. First, it explored several strategies for observing and controlling the reactive execution of aggregate specifications, leading to a step-by-step reactive execution model. Then, it defined a concrete solution for evaluating convergence-based spatio-temporal properties of FRASP systems through simulation. Finally, it developed an extensive test suite, providing valuable insights on the current state of the DSL and future challenges to overcome.

The results proved the overall soundness of FRASP, consolidating its foundations in support of upcoming improvements and extensions. Some issues were identified and analyzed, including a limitation in referencing the computation of aggregate specifications within other specifications, and a scheduling problem with the previous and current simulators, causing inconsistencies during the evolution of aggregates over time.

The contributions of this project include a modular implementation of several simulators, a collection of operators for the analysis and monitoring of reactive variables, and explicit support for dynamic environments and proper sensors. Unfortunately, while the solution was designed for concurrency, the strong consistency of the underlying reactive engine proved to negate parallelism, limiting the benefits of concurrency. On the one hand, there is opportunity for reducing the complexity of the current design by giving up concurrency, possibly embracing a complete FRP implementation without compromises; on the other hand, some questions arise about the implications of such constraints on the deployment of aggregate specifications in real-world distributed systems, specifically collective adaptive systems, hinting towards the necessity for further research into distributed reactive solutions.

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.

[ACDV23] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. Computation against a neighbour: Addressing large-scale distribution and adaptivity with functional programming and scala. *Logical Methods in Computer Science*, Volume 19, Issue 1, jan 2023.

[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.

[Art] Artima. ScalaTest documentation. `https://www.scalatest.org/`. Accessed: 02-25-2024.

[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, jul 2016.

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, 2015.

[Cas23a] Roberto Casadei. Artificial collective intelligence engineering: A survey of concepts and perspectives. *Artificial Life*, 29(4):433–467, nov 2023.

[Cas23b] Roberto Casadei. Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling. *ACM Comput. Surv.*, 55(13s), jul 2023.

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

[CAV21] Roberto Casadei, Gianluca Aguzzi, and Mirko Viroli. A programming approach to collective autonomy. *Journal of Sensor and Actuator Networks*, 10(2), 2021.

[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*, mar 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.

[Osh13] Roy Osherove. *The Art of Unit Testing*. Manning, second edition, nov 2013.

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