ScaFi-Web: a Web-based Application for Field-based Coordination Programming

Gianluca Aguzzi, Roberto Casadei[0000−0001−9149−949X], Niccolò Maltoni, Danilo Pianini[0000−0002−8392−5409], and Mirko Viroli[0000−0003−2702−5702]

Alma Mater Studiorum–Università di Bologna, Italy
{gianluca.aguzzi, roby.casadei, niccolo.maltoni, danilo.pianini, mirko.viroli}@unibo.it

Abstract. Field-based coordination is a model for expressing the coordination logic of large-scale adaptive systems, composing functional blocks from a global perspective. As for any coordination model, a proper toolchain must be developed to support its adoption across all development phases. Under this point of view, the ScaFi toolkit provides a coordination language (field calculus) as a DSL internal in the Scala language, a library of reusable building blocks, and an infrastructure for simulation of distributed deployments. In this work, we enrich such a toolchain by introducing ScaFi-Web, a web-based application allowing in-browser editing, execution, and visualisation of ScaFi programs. ScaFi-Web facilitates access to the ScaFi coordination technology by flattening the learning curve and simplifying configuration and requirements, thus promoting agile prototyping of field-based coordination specifications. In turn, this opens the door to easier demonstrations and experimentation, and also constitutes a stepping stone towards monitoring and control of simulated/deployed systems.

Keywords: Field-based coordination · Aggregate Programming · Online Playground.

1 Introduction

Emerging trends such as the Internet of Things (IoT) and Cyber-Physical Systems (CPS) foster a vision of large-scale coordinated systems of situated devices that operate in a dynamic environment and seamlessly collaborate to reach global goals. When designing such systems, one key engineering challenge is mapping the intended global and adaptive behaviour that the system should exhibit to the local behaviour and interaction of its components. To tackle this issue, research often looked at nature for inspiration: (bio)chemistry [35], ecosystems [37], ethology [31], and fields in physics [25,26]. Accordingly, several approaches in the literature proposed the idea of programming coordination and adaptation through reusable abstractions [13,21] and interaction structures [24,1,23]. Recently, *aggregate computing* [8] has emerged as a paradigm, stemming from *field-based coordination* [34], fostering a top-down and global-to-local approach to

the specification of the collective adaptive behaviour of a system. The idea is to exploit the *(computational) field* abstraction – a "collective" data structure mapping any device of the system to a corresponding value – to program the system as a whole; namely, the program expressing the behaviour of the collective takes the form of a composition of functions from input fields (*e.g.*, sensor fields) to output fields (*e.g.*, value and actuation fields) [5]. Aggregate programming is supported by several various domain-specific languages (DSL), including Proto [7], a Scheme-based DSL; Protelis [29], a Java-interoperable stand-alone DSL; FCPP [2], a lightweight native implementation; and ScaFi [19], a modern Scala-based DSL.

Language-based approaches often enjoy solid formal foundations and demonstrable properties [34], but in turn require the user a potentially steep learning curve which includes learning a paradigm, a language, and a development toolset. In the case of languages for distributed systems, a further element of complexity exists: programs should be executed on a *network of devices*. Due to the costs and impracticality of real deployments, a typical approach for studying, testing, and developing applications with distributed programming languages consists in using *simulators*. However, this introduces further practical issues to the setup of a development environment; such accidental complexity could hinder accessibility.

Along the line of pioneering platforms such as Web Proto [32], this paper presents ScaFi-Web, an online playground for the ScaFi aggregate programming DSL, featuring an intuitive web-based graphical user interface (GUI), an integrated simulation environment with zero-configuration overhead from the user, and a guided tour of the key language features. The paper continues as follows: Section 2 discusses motivation and related work; Section 3 provides background on ScaFi and field calculus; Section 4 covers the ScaFi-Web platform (the tool at the core of this contribution) and its use cases; finally, Section 5 presents the conclusions and points out directions for future work.

2 Motivation and Related Work

Learning a new language may not be trivial, especially when it involves learning new paradigms and/or new development tools. The literature on learning computational concepts suggests that two key elements for effectively introducing novel programming concepts are *simplicity* and *visibility* [12], intended, respectively, as the number of interacting parts that should be understood to realise how the system works, and as the possibility to isolate and inspect such parts. Also, in [11] four critical dimensions are identified:

1. *orientation*, namely finding out what a programming language is for and which class of problems it applies well to;
2. *understanding of the notional machine*, namely understanding how changes in the program affect results;
3. *notation*, namely getting acquainted with the syntax and the underlying semantics; and finally

4. *pragmatics*, including all skills related to assemble and use an environment supporting development, testing, and debugging.

To tackle the problem, it is common practice to provide graphical tools that enable the user to kickstart a project. Typically, however, these tools have prerequisites (runtimes, editors, integrated development environments, plugins, custom software modules) that require an investment of time and effort to begin the learning process. This initial investment, usually easily tolerated by the experienced practitioner or the committed learner, could be costly enough to discourage occasional users from experimenting with a novel tool or language. For this reason, all major modern general-purpose languages have adopted the strategy of providing web-based programming playgrounds with the sole prerequisite of a modern web browser, using no third-party tool at all. A purely browser-based tool has several advantages over a classic application:

- *portability* – experimentation can happen from any device equipped with a modern browser, thus including devices usually not supported by classic development stacks (gaming consoles, smart TVs, set-top boxes, and the like), as well as niche operating systems;
- *future-proofness* – any future software environment with a standard-compliant web browser will be able to run the application;
- *zero-permission* – the absence of any part to be locally installed implies that the experimentation can happen entirely in userspace, while very often a correct local installation of a runtime environment should be performed by a system administrator;
- *zero-time experimentation* – executing a simple experiment is as easy as opening a browser tab and typing the program, as opposed to manually using compilers or starting a stand-alone application.

Well-designed web playgrounds hide many of the issues related to the *pragmatics* of learning a language, postponing them and hence allowing developers to focus on other dimensions first. Also, by presenting a carefully designed learning path of increasingly complex exercises and examples, these platforms increase *simplicity* and favour the *understanding of the notional machine*, ultimately promoting *self-orientation* as well. Examples of web playgrounds for modern programming languages include Kotlin Koans[1], ScalaFiddle[2], and Rust Playground[3].

Web Proto [32] pioneered the approach for languages dedicated to distributed systems by providing:

- an online editor,
- an interpreter for the Proto [7] language for spatial computing based on emscripten[4],

[1] `https://play.kotlinlang.org/koans`
[2] `https://scalafiddle.io/`
[3] `https://play.rust-lang.org/`
[4] `https://emscripten.org/`

– an in-browser simulated environment, and
– the possibility to share code between different users.

Unfortunately though, Proto is no longer maintained (as it has been replaced by Protelis [29][5], which has no web environment at the time of writing); and Web Proto, to the best of our knowledge, is no longer reachable, and there is no plan to deploy it again. The approach proposed in Web Proto inspired the tool at the centre of this paper: ScaFi-Web, which directly supports modern aggregate programming tools like ScaFi (see Section 3), provides a cleaner and clever web-based technical solution (see Section 4), and has well-defined pedagogical use cases (see Section 4.2). Unlike Web Proto, which is a JavaScript re-implementation of Proto, this project fully reuses the mainline ScaFi code through transpilation into a browser-compatible target, considerably reducing maintenance and enabling feature parity with the classic version. Indeed, the DSL and the simulator provided by ScaFi have been successfully brought into the JavaScript world leveraging Scala.js, with no special changes or disruptive adaptations of the original code. Furthermore, while Web Proto was designed to support only one language, several ScaFi-Web components (primarily, the visualisation section) are agnostic to the aggregate language of choice, in the spirit of supporting different languages and platforms (distributed or simulated) in the future.

3 The ScaFi Aggregate Programming Toolchain

ScaFi[6] (SCAla FIeld) is a modular Scala-based toolchain for aggregate computing. It provides *(i)* a DSL implementation of the field calculus [34,19] designed to simplify embedding in common general-purpose languages; *(ii)* a library of reusable aggregate behaviour functions; *(iii)* simulation support for logical networked systems driven by ScaFi programs (*aggregate systems*), through an internal engine or a plug-in for the Alchemist simulator [28,36]; and *(iv)* an actor-based platform [18] to deploy aggregate systems in real-world clusters. ScaFi has been used as a framework for experimenting with new field calculus constructs [20] and for building decentralised algorithms and applications in large-scale computing scenarios ranging from trust-based collaborative systems [16] to resource coordination in IoT and edge computing [17]. In a typical aggregate application development workflow, a developer iteratively constructs a ScaFi program, performs a set of simulations and tests to evaluate how the system performs in a range of target environments, iterates until sufficient confidence on correctness is achieved, and finally deploys the program to a set of nodes running an aggregate computing middleware (such as the ScaFi actor-based platform), when further maintenance, monitoring, and control is generally required. ScaFi, with respect to other aggregate programming languages, benefits from being an internal DSL, enabling reuse of the toolchain available for its host language (Scala).

[5] `https://bit.ly/3uRZYgp`
[6] `https://scafi.github.io`

3.1 Field calculus and the ScaFi DSL

Field calculus The field calculus [5] is a core language that captures the essential aspects for programming self-organising systems. In this language, programs – also called *field programs* or *aggregate programs* – consist of expressions that conceptually manipulate distributed state. The distributed state is modelled by the *computational field* abstraction, which is essentially a map from any device of a system domain to computational values. For instance, globally querying a "temperature sensor" in a sensor network would yield a field of temperatures, which maps each device with the corresponding temperature that it read. The field calculus is based on a minimal set of operators such as the following.

- *Stateful field evolution*: expression `rep(`e_1`){(x) =>`e_2`}` describes a field evolving in time. e_1 imposes the initial field value, and function `(x) =>` e_2 declares how the field changes at each execution.
- *Neighbour interaction*: expression `nbr{e}` builds a *neighbouring field*, a view of the field values in the surroundings of each device where neighbours are mapped to their evaluations of `e`.
- *Domain partitioning*: expression `if(`e_0`){`e_1`}{`e_2`}` splits the computational field into two non-communicating zones hosting isolated subcomputations: e_1 where e_0 is `true`, and e_2 where e_0 is `false`.

System model and execution model The logical structure of an aggregate system merely consists of a network of nodes connected through a *neighbouring relationship*. The network semantics of the field calculus [5] defines what local execution protocol any device belonging to the aggregate system should follow so that an aggregate program leads to the designed collective behaviour. The basic idea is that any device should "continuously" sense, coordinate, and act over its local context. Therefore, every device performs *asynchronous rounds of execution*, where each round consists of the following steps:

1. *context gathering*—the device retrieves sensor data, messages from neighbours, and its previous state;
2. *computation*—the device evaluates the aggregate program, which yields an output as well as a coordination message – called an *export* – meant to be shared with neighbours;
3. *context action*—the device runs actuations and broadcasts the export to its neighbours.

Following this protocol, a collective execution of a field program can lead to self-organising behaviour, in a powerful, emergent way—cf. the channel example in Section 3.2.

The ScaFi aggregate programming language The ScaFi DSL implements a field calculus variant, called FScaFi [19], in which `nbr` does not directly yield a computational field but rather must be evaluated while folding over neighbourhoods (through a function `foldhood`, described next). The core language constructs are captured in Scala through the following trait:

```
trait Constructs {
  def nbr[A](expr: => A): A
  def rep[A](init: =>A)(fun: (A) => A): A
  def foldhood[A](init: => A)(aggr: (A, A) => A)(expr: => A): A
  // Contextual, but foundational
  def mid(): ID
  def sense[A](name: LSNS): A
  def nbrvar[A](name: NSNS): A
}
```

In particular, `mid` is a built-in sensor that provides the identifier (ID) of the running node, whereas `sense` and `nbrvar` are two operators included for interacting with the environment. The former abstracts a local sensor queries (*e.g.*, the value of a temperature sensor), and the latter instead produces a field of local sensor readings that are *relative to* neighbours. A typical example of `nbrvar` is `nbrRange`, an "environmental sensor" that creates a field of distances mapping every neighbour with its relative distance from the running node. Moreover, the field calculus `if` expression became `branch` in Scala, because `if` is a reserved keyword. Now, we provide a brief tutorial on ScaFi. As a very trivial example, a local value such as

```
true
```

can be used to represent a uniform and constant field in the system (indeed, any node evaluates such expression to `true`). Using `rep`, we can evolve state, *e.g.*, to implement a local round counter:

```
rep(0){ _ + 1 } // _ + 1 is a Scala shorthand for function x => x + 1
```

This code snippet describes a field that starts from 0 and then evolves by increasing by one. To pass from purely local to global computations, we need devices to interact. This is possible by combining the `nbr` and `foldhood` operators:

```
foldhood(mid()) { Math.min } { nbr(mid()) }
```

The `foldhood` operator accepts three arguments: *(i)* an initial value *(ii)* an accumulator, i.e. a strategy used to combine a neighbour's value to the partial accumulation, and *(iii)* an expression to be evaluated against any neighbour. In this example, we gather the neighbours' IDs, folding over them to take the minimum ID in the neighbourhood. We can combine `rep` and `foldhood` to build fields progressively constructed and refined by coordination of all the nodes. For instance, expression

```
rep(mid()){ minId => foldhood(minId) { Math.min } { nbr(mid()) } }
```

represents a gossip process that will produce a field eventually tending to the minimum ID of the whole system.

The key idea of aggregate computing is to find recurrent patterns and then use the functional abstraction to create reusable building blocks [33]. One of

these is `G`, which generalises *gradient* computations [4] to allow propagation of information form a node outwards:

```
G[F](source: Boolean, field: F, accumulator: F => F, metric: Metric)
```

`G` diffuses information by accumulating `field` data while ascending a potential field centred where `source` is `true`. Graphically, Figure 1 shows the operator behaviour. On top of `G`, we can build other significant functions as `distanceTo` (i.e., a self-healing version of the Bellman-Ford algorithm [27]),

```
def distanceTo(source: Boolean, metric: Metric = nbrRange): Double = {
  val field = if(source){ 0.0 } { Double.PositiveInfinity }
  G(source, field, _ + metric(), metric)
}
```

and `distanceBeetwen` (i.e. the network-propagated self-healing minimum distance between source and target nodes):

```
def distanceBetween(source: Boolean,target: Boolean,
  metric: Metric = nbrRange
): Double =
  G(source, distanceTo(target, metric), v => v, metric = metric)
```

Finally, using only this subset of operators, it is possible to build non-trivial examples, such as the self-healing channel explained below.

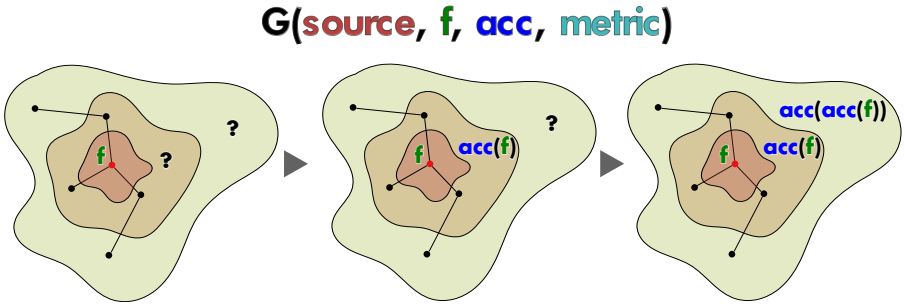

Fig. 1. Evolution in time of `G`. Red color marks the source node. The links express a neighbour relation between nodes. The time flows from the left to the right. Initially, the source node computes its value as `f` (initial value). The potential field will be created following the metric passed. Step by step, `f` will be shared through the accumulator function.

3.2 The Channel Example

A paradigmatic example of aggregate computing is the *channel*, an algorithm yielding a self-healing Boolean field that is (eventually) `true` along the minimal

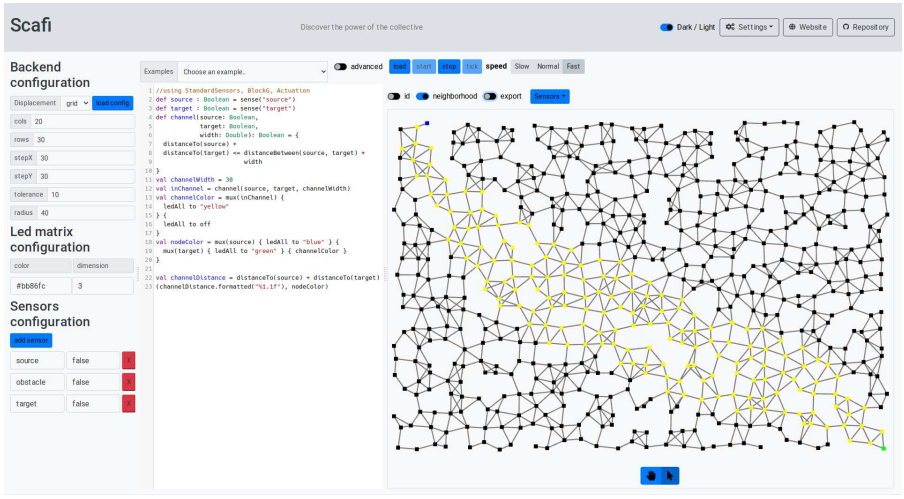

Fig. 2. ScaFi-Web (`https://scafi.github.io/web`) running the channel example.

hop-by-hop path from a `source` to a `target` device (as identified also by Boolean fields). Self-healing field means that the channel structure recovers itself after failures, without human interaction. This can be used *e.g.* for navigating people or streaming information towards target locations. An implementation leverages distance estimations (based on gradients [4]) and the triangular inequality:

```
distanceTo(source) + distanceTo(target) <= distanceBetween(source,target)
```

Changes to the inputs (`source` and `target` fields, topology, neighbours' messages with distance estimates) affect local outputs, which then affect neighbours', and ultimately the global response of the system—effectively steering the emergent, *self-organising* (i.e. global patterns emerge from local node's interactions) behaviour of the aggregate. More details on this example can be found in [6]. Figure 2 shows a snapshot of a channel simulation in ScaFi-Web.

3.3 ScaFi Programming in Practice

Programming in ScaFi requires building a model of the system, writing a field-based program assuming that model, and finally deploying and running the corresponding application. However, setting up a (simulated) system requires familiarity with a set of tools and notions, including: build automation and dependency management tools (such as SBT or Gradle) to import ScaFi modules as dependencies, integrated-development environments (such as IntelliJ Idea) for editing programs, library and framework APIs for configuring and integrating several components. In particular, some boilerplate code or configuration files may be needed: the program expressing the system behaviour, the structure of

the simulated system, the dynamics of the simulated system (cf. scheduling, failure injection, interaction) and the environment, simulation parameters (*e.g.*, for reproducibility and control of scenarios), as well as inspection and visualisation aspects (*e.g.*, of the graphical evolution of the system and the data produced by its components).

Therefore, any support reducing the gap from the aggregate specification to a graphical representation of a running system can be useful to promote accessibility, learning, and experimentation—hence motivating ScaFi-Web.

4 ScaFi-Web

ScaFi-Web[7] is an online playground for learning the ScaFi toolkit, experimenting with it, and monitoring executions in a browser. It features:

- an interactive editor for writing ScaFi programs;
- a guided tour of the most prominent features, kickstarting development;
- a set of increasingly complex examples;
- an in-browser simulated network of devices hosting the execution;
- visualisation, inspection, and interaction tools integrated with the simulated environment.

Besides flattening the learning curve of a novel paradigm and allowing first-hand experimentation with zero configuration, ScaFi-Web also provides a stepping stone towards a monitoring and control system for aggregate computing deployments. In the following sections, we explain the architecture in detail, motivating design choices and exposing some relevant use cases and opportunities.

4.1 Architecture details

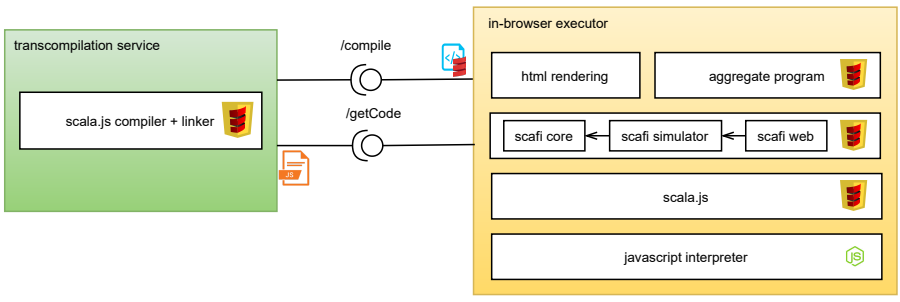

Fig. 3. Architecture of ScaFi-Web, with the involved technologies.

[7] `https://scafi.github.io/web`

ScaFi-Web finds its novelty in the capability of running the whole aggregate program inside the runtime of the browser. Most modern browsers can execute a limited number of code targets, mostly JavaScript and WebAssembly (WASM) [30]. Scala, the language hosting the ScaFi DSL, primarily targets the Java Virtual Machine (JVM), whose execution on browser platform was deprecated[8] years ago and is no longer supported.

To be able to execute ScaFi code (and thus Scala code) client-locally, a transcompilation from Scala to one of the aforementioned languages is required; JavaScript has been selected as the only web target currently supported by the Scala native compiler[9]. User-written Scala code can be transcompiled into JavaScript on-the-fly and injected into the simulator, leveraging the `scalac` compiler with Scala.js [22]. The Scala compiler, however, currently requires to be executed in a JVM, and hence off-browser (despite being written mostly in Scala, there is no JavaScript version of `scalac` at the time of writing). Consequently, the ScaFi-Web architecture has been designed with two components: a frontend hosting the interpreter, simulator, and user interface; and a transcompilation service, in charge of producing JavaScript code for the aggregate specifications written in Scala.

Transcompilation service The remote service instance, depicted on the left of Figure 3, exposes the transcompilation service via RESTful HTTP APIs. The route `/compile` accepts POST requests containing the ScaFi code to be transcompiled. Leveraging the Scala compiler, it creates a REST resource identified by a UUID with the transpiled JavaScript code. The identifier returned to the client can be used on the `/getCode/{UUID}` API via GET requests to download the transpiled code.

The service is platform-agnostic and can be executed on all operating systems and runtimes that can host a JVM. To simplify the deployment of ScaFi-Web instances, the service has been made publicly available[10] in a containerised [14] fashion as a Docker image, deployable on any compatible container runtime.

Client interface Client-side, a Single-Page Application (SPA) was implemented to manage *(i)* programs execution, *(ii)* simulation management, and *(iii)* page rendering. The interface, as visible in Figure 2, is structured in three parts, each exposing a different logical control.

On the left, a configuration panel enables control of the network shape and device sensors, enabling users to design their own deployment configurations. Moving on the central part, an editor is available to fiddle with the provided examples or write fresh new code. Editing can be performed in two flavours: *simplified* (selected by default), or *advanced*.

[8] `http://openjdk.java.net/jeps/289`
[9] `https://archive.is/SaV6B`
[10] `https://hub.docker.com/r/gianlucaaguzzi/scafi-web`

In simplified mode, inspired by the interaction typical of REPL interpreters [10], the editor hides all the boilerplate code, allowing for a very straightforward hands-on with the core language mechanisms. In advanced mode, instead, these details are exposed, bridging the gap towards full-fledged development environments.

Finally, the program can be executed with the controls available on the rightmost section of the page. Pressing the play button causes the application to transpile under-the-hood the code in the editor (by leveraging the aforementioned remote service), injecting the resulting JavaScript in the browser-hosted ScaFi simulator.

4.2 Usage scenarios

Learning and education Web-based playgrounds are a trending way to experiment with languages, they are well accepted as they lower the adoption and learning curves. ScaFi-Web guides the user to an understanding of field-based coordination by *(i)* exposing an environment with minimal requirements (a modern browser); *(ii)* including a tour of its functions; *(iii)* presenting a sequence of guided examples of increasing complexity that can be simulated immediately; *(iv)* providing simplified access to the simulated sensors and actuators (*e.g.* for controlling movement and colour of devices).

Fast Prototyping Programming complex coordination logic is challenging, even with paradigms that promote collective behaviour abstractions to first-class: robust specifications usually result from an iterative, incremental process where ScaFi programs are progressively refined. Simulation has a key role in this development workflow, as it allows for observing and controlling the software in a variety of scenarios (different network structures, dynamics, and perturbations) without the issues related to actual deployment. ScaFi-Web supports this kind of workflow, by providing an out-of-the-box web-based simulation environment with zero installation overhead.

4.3 Roadmap to monitoring and control of deployed systems

The monitoring and control of distributed systems is a prominent practical issue. In the context of field-based coordination, automated runtime verification approaches have been recently investigated, whereby spatial or temporal logics are mapped to field calculus programs to directly encode the behaviour of decentralised monitors [3]. However, these techniques are complementary to monitoring and control activities carried out by humans, which may need remote frontends to inspect and act over a running system. The ScaFi-Web's frontend has been designed to be adaptable to different backends; indeed, the UI is completely separated from the underlying aggregate execution system (called `Support` in ScaFi-Web). We plan to evolve ScaFi-Web into a platform for remote monitoring and control of aggregate systems. In particular, we aim to:

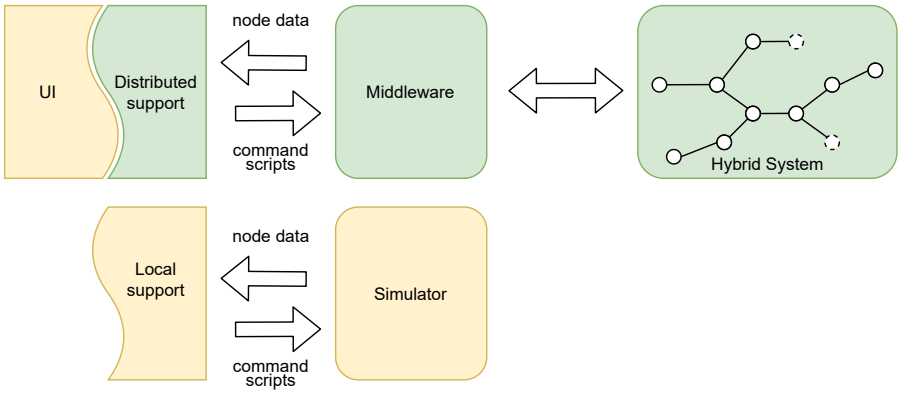

Fig. 4. This figure shows the logical components currently available (yellow) and those that have to be developed to support a full-fledged monitoring and control solution. Notice the separation between UI and support. (Notation: nodes with dotted border are purely simulated.)

1. define a middleware in charge of both retrieving values from the different nodes in the system, injecting aggregate code and sending well-defined commands;
2. create a new `Support` able to communicate with the aforementioned middleware (*e.g.* via WebSocket);
3. create a `Support`-to-UI component that understands and manages the aggregate computing languages of the new `Support`;
4. establish a simulation-to-middleware component that can inspect the overall system status;
5. realise tests upon real systems;
6. introduce the opportunity to orchestrate hybrid real-virtual systems, in which real devices can interact with virtual devices managed by a simulation platform.

Figure 4 summarises the general idea of the final product. With the proposed architecture, it will be possible to inject new behaviours by either specific commands (*e.g.* move the alpha node to X,y) or by injecting new aggregate programs (*e.g.* to verify a property at runtime, as in [3]). Finally, given the hybrid nature of the system, it can be interesting to spawn new simulated nodes at runtime. This can be useful, for instance, to improve the performance of programs performing density-sensitive operations [9] when executing in low-device-density conditions.

5 Conclusion and Future Work

In this paper, we have presented ScaFi-Web, a web-based playground and frontend for simulated aggregate computing systems, enabling seamless and uni-

versal access to the ScaFi aggregate programming toolchain. It provides an environment with zero-installation overhead and pedagogical support for learning, exploratory testing, and easy application deployment.

As a future work, we would like to extend ScaFi-Web with out-of-the-box support for monitoring and control of deployed systems. Additionally, it would be nice to provide a graphical DSL allowing the creation of aggregate specifications by the composition of algorithmic blocks, hence simplifying application development.

References

1. Arbab, F.: A behavioral model for composition of software components. Obj. Logiciel Base données Réseaux **12**(1), 33–76 (2006). https://doi.org/10.3166/objet.12.1.33-76
2. Audrito, G.: FCPP: an efficient and extensible field calculus framework. In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Washington, DC, USA, August 17-21, 2020. pp. 153–159. IEEE (2020). https://doi.org/10.1109/ACSOS49614.2020.00037
3. Audrito, G., Casadei, R., Damiani, F., Stolz, V., Viroli, M.: Adaptive distributed monitors of spatial properties for cyber–physical systems. Journal of Systems and Software **175**, 110908 (May 2021). https://doi.org/10.1016/j.jss.2021.110908
4. Audrito, G., Casadei, R., Damiani, F., Viroli, M.: Compositional blocks for optimal self-healing gradients. In: 11th IEEE SASO 2017. pp. 91–100. IEEE Computer Society (2017). https://doi.org/10.1109/SASO.2017.18
5. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Trans. Comput. Log. **20**(1), 5:1–5:55 (2019). https://doi.org/10.1145/3285956
6. Bachrach, J., Beal, J., McLurkin, J.: Composable continuous-space programs for robotic swarms. Neural Comput. Appl. **19**(6), 825–847 (2010). https://doi.org/10.1007/s00521-010-0382-8, `https://doi.org/10.1007/s00521-010-0382-8`
7. Beal, J., Bachrach, J.: Infrastructure for engineered emergence on sensor/actuator networks. IEEE Intell. Syst. **21**(2), 10–19 (2006). https://doi.org/10.1109/MIS.2006.29
8. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the internet of things. Computer **48**(9), 22–30 (2015). https://doi.org/10.1109/MC.2015.261
9. Beal, J., Viroli, M., Pianini, D., Damiani, F.: Self-adaptation to device distribution in the Internet of Things. ACM Transaction on Autonomous and Adaptive Systems **12**(3), 12:1–12:29 (Sep 2017). https://doi.org/10.1145/3105758
10. van Binsbergen, L.T., Merino, M.V., Jeanjean, P., van der Storm, T., Combemale, B., Barais, O.: A principled approach to REPL interpreters. In: Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. ACM (Nov 2020). https://doi.org/10.1145/3426428.3426917
11. du Boulay, B.: Some difficulties of learning to program. J. Educ. Comput. Res. **2**(1), 57–73 (Feb 1986). https://doi.org/10.2190/3lfx-9rrf-67t8-uvk9
12. du Boulay, B., O'Shea, T., Monk, J.: The black box inside the glass box: presenting computing concepts to novices. Int. J. Hum. Comput. Stud. **51**(2), 265–277 (1999). https://doi.org/10.1006/ijhc.1981.0309

13. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECO: an ensemble-based component system. In: CBSE'13. pp. 81–90. ACM (2013). https://doi.org/10.1145/2465449.2465462
14. Butzin, B., Golatowski, F., Timmermann, D.: Microservices approach for the internet of things. In: 21st IEEE ETFA 2016. pp. 1–6. IEEE (2016). https://doi.org/10.1109/ETFA.2016.7733707
15. Casadei, R., Aguzzi, G., Peruzzi, M., Maltoni, N., Viroli, M.: scafi/scafi-web: Scafi web 0.1.0 alpha (2021). https://doi.org/10.5281/ZENODO.4688017, `https://zenodo.org/record/4688017`
16. Casadei, R., Aldini, A., Viroli, M.: Towards attack-resistant aggregate computing using trust mechanisms. Science of Computer Programming (2018). https://doi.org/10.1016/j.scico.2018.07.006
17. Casadei, R., Tsigkanos, C., Viroli, M., Dustdar, S.: Engineering resilient collaborative edge-enabled iot. In: 2019 IEEE International Conference on Services Computing (SCC). pp. 36–45 (July 2019). https://doi.org/10.1109/SCC.2019.00019
18. Casadei, R., Viroli, M.: Programming actor-based collective adaptive systems. In: Programming with Actors: State-of-the-Art and Research Perspectives, LNCS, vol. 10789, pp. 94–122. Springer International Publishing (2018). https://doi.org/10.1007/978-3-030-00302-9_4
19. Casadei, R., Viroli, M., Audrito, G., Damiani, F.: Fscafi : A core calculus for collective adaptive systems programming. In: ISoLA (2). LNCS, vol. 12477, pp. 344–360. Springer (2020)
20. Casadei, R., Viroli, M., Audrito, G., Pianini, D., Damiani, F.: Engineering collective intelligence at the edge with aggregate processes. Eng. Appl. Artif. Intell. **97**, 104081 (2021). https://doi.org/10.1016/j.engappai.2020.104081
21. Denti, E., Natali, A., Omicini, A.: Programmable coordination media. In: Coordination Languages and Models, Second International Conference, COORDINATION '97, Berlin, Germany, September 1-3, 1997, Proceedings. LNCS, vol. 1282, pp. 274–288. Springer (1997). https://doi.org/10.1007/3-540-63383-9_86
22. Doeraene, S.: Cross-platform language design in scala.js (keynote). In: Erdweg, S., d. S. Oliveira, B.C. (eds.) Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, MO, USA, September 28, 2018. p. 1. ACM (2018). https://doi.org/10.1145/3241653.3266230, `https://doi.org/10.1145/3241653.3266230`
23. Fernandez-Marquez, J.L., Serugendo, G.D.M., Montagna, S., Viroli, M., Arcos, J.L.: Description and composition of bio-inspired design patterns: a complete overview. Nat. Comput. **12**, 43–67 (2013). https://doi.org/10.1007/s11047-012-9324-y
24. Frey, S., Diaconescu, A., Menga, D., Demeure, I.: A holonic control architecture for a heterogeneous multi-objective smart micro-grid. In: 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems. IEEE (Sep 2013). https://doi.org/10.1109/saso.2013.11
25. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications with the TOTA middleware. In: PerCom 2004. pp. 263–276. IEEE Computer Society (2004). https://doi.org/10.1109/PERCOM.2004.1276864
26. Mamei, M., Zambonelli, F., Leonardi, L.: Co-fields: Towards a unifying approach to the engineering of swarm intelligent systems. In: ESAW 2002. LNCS, vol. 2577, pp. 68–81. Springer (2002). https://doi.org/10.1007/3-540-39173-8_6
27. Mo, Y., Dasgupta, S., Beal, J.: Robustness of the adaptive bellman -ford algorithm: Global stability and ultimate bounds. IEEE Trans. Autom. Control. **64**(10),

4121–4136 (2019). https://doi.org/10.1109/TAC.2019.2904239, https://doi.org/10.1109/TAC.2019.2904239
28. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with Alchemist. Journal of Simulation (2013). https://doi.org/10.1057/jos.2012.27
29. Pianini, D., Viroli, M., Beal, J.: Protelis: practical aggregate programming. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015. pp. 1846–1853. ACM (2015). https://doi.org/10.1145/2695664.2695913
30. Rossberg, A., Titzer, B.L., Haas, A., Schuff, D.L., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F., Holman, M.: Bringing the web up to speed with webassembly. Commun. ACM **61**(12), 107–115 (2018). https://doi.org/10.1145/3282510, https://doi.org/10.1145/3282510
31. Trianni, V., Nolfi, S., Dorigo, M.: Evolution, self-organization and swarm robotics. In: Natural Computing Series, pp. 163–191. Springer Berlin Heidelberg (2008). https://doi.org/10.1007/978-3-540-74089-6_5
32. Usbeck, K., Beal, J.: Web proto: Aggregate programming for everyone. In: 7th IEEE SASOW, 2013. pp. 17–18. IEEE Computer Society (2013). https://doi.org/10.1109/SASOW.2013.12
33. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans. Model. Comput. Simul. **28**(2), 16:1–16:28 (Mar 2018)
34. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From distributed coordination to field calculus and aggregate computing. J. Log. Algebraic Methods Program. **109** (2019). https://doi.org/10.1016/j.jlamp.2019.100486
35. Viroli, M., Casadei, M.: Biochemical tuple spaces for self-organising coordination. In: LNCS, pp. 143–162. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-642-02053-7_8
36. Viroli, M., Casadei, R., Pianini, D.: Simulating large-scale aggregate mass with alchemist and scala. In: Proceedings of FedCSIS 2016. Annals of Computer Science and Information Systems, vol. 8, pp. 1495–1504. IEEE (2016). https://doi.org/10.15439/2016F407
37. Zambonelli, F., Omicini, A., Anzengruber, B., Castelli, G., Angelis, F.L.D., Serugendo, G.D.M., Dobson, S.A., Fernandez-Marquez, J.L., Ferscha, A., Mamei, M., Mariani, S., Molesini, A., Montagna, S., Nieminen, J., Pianini, D., Risoldi, M., Rosi, A., Stevenson, G., Viroli, M., Ye, J.: Developing pervasive multi-agent systems with nature-inspired coordination. Pervasive Mob. Comput. **17**, 236–252 (2015). https://doi.org/10.1016/j.pmcj.2014.12.002