FCPP: an efficient and extensible Field Calculus framework

Giorgio Audrito
Computer Science Department and C3S
University of Torino, Torino, Italy
email: giorgio.audrito@unito.it ORCID: 0000-0002-2319-0375

***Abstract*—The Field Calculus is a promising language for the self-organisation of distributed devices, allowing to express on a high level of abstraction complex distributed algorithms with robust behaviour guarantees. This language has been argued to be fruitfully applicable to many different contexts: wireless sensor networks, internet of things, self-organising edge, fog or cloud computing scenarios, and simulations of such. However, existing implementations of this language rely on the Java Virtual Machine and have an high performance overhead, impairing their usability in contexts where performance is critical (cloud) or computational resources are tightly bounded (WSN/IoT).**

In this paper we present FCPP, a novel implementation of the Field Calculus as a C++ library. The library is built as a component-based system, in order to be easily extensible to fit different contexts. Furthermore, it leverages C++ template patterns to allow compile-time optimisation and minimal performance overhead, and enables fine-grained parallelism for scalability in self-organising cloud applications. A case study of an edge simulation shows the performance improvement compared to existing Field Calculus implementations, while preserving the same level of abstraction. This translates to a significant speed-up in the development process of distributed algorithms, paving the way towards application scenarios for which existing tools are unsuitable: microcontroller systems and self-organising cloud.

***Index Terms*—edge computing, aggregate computing, programming languages, field calculus, toolchains**

I. Introduction

As the density of computing devices in human environments is continuing to increase, the problem of coordinating their behaviour is correspondingly growing in importance. Over the past few decades, humanity moved from many people sharing a single computer to several computing devices per person: phones, watches, vehicles and much more are becoming capable of computing, sensing and communicating. The growing numbers of devices involved translate into rising costs of maintaining distributed systems, making the deployment of global-level software services by individual programming of every single node more and more problematic, and driving a search for solutions increasing the autonomy of computing systems while reducing their overall complexity.

Several macro-programming and spatial self-organisation approaches are being investigated [1]–[3], aiming at the abstraction of a network as a single spatially distributed and inherently robust platform to be programmed as a whole. Aggregate programming [4], [5] is an emerging approach among them, based on the functional composition of reusable blocks of collective behaviour, with the goal of effectively achieving resilient complex behaviours in dynamic networks. This feature allows to study robustness properties like self-stabilisation [6] and density independence [7] by first proving them on simple blocks (e.g., broadcast [6], distance estimation [8]–[10], data aggregation [11]) and then transferring them to more complex systems built on top by functional composition.

Aggregate programming is built on the Field Calculus [12], a minimal functional programming language for the specification and composition of collective behaviours, with sufficient power to express any distributed computation [13], and built-in global-to-local translation formalised as equivalent local and aggregate semantics. Layers of increasingly complex self-organisation routines can be defined on top of it, simplifying the engineering of robust distributed systems in contexts such as smart-cities, robot/drone swarms, and tactical networks (see [5] for an extended presentation of aggregate programming).

Several tools have been developed for supporting the execution of field calculus-based algorithms: Proto programming language and simulation engine [14] (discontinued), Alchemist[1] simulator [15] with the Protelis[2] language [16], and finally the Scafi[3] programming language and simulation engine [17]. The existing and supported tools already cover a number of application scenarios, however, they all rely on the Java Virtual Machine (JVM) for execution, limiting their applicability to platforms (microcontrollers) where it may not be available. Furthermore, they are not optimised for performance and do not support fine-grained parallelism, reducing their effectiveness for self-organising cloud scenarios.

In this paper we present FCPP,[4] a novel implementation of the Field Calculus as a C++14 library, together with its own performance-oriented simulation engine. A layered component-based architecture ensures convenient extensibility of the tool, which thanks to the low system requirements and the widespread support for C++ on architectures, could potentially be deployed on systems of any sort (including microcontrollers). Furthermore, a built-in support for fine-grained parallelism paves the way towards future extensions of the tool supporting self-organising cloud scenarios. At the current state, the performance improvements with respect to

[1]Available at http://alchemistsimulator.github.io.
[2]Available at http://protelis.github.io.
[3]Available at https://scafi.github.io.
[4]Available at https://fcpp.github.io.

existing tools already allow for more efficient simulation of aggregate algorithms, speeding up their development process.

The reminder of this paper is organised as follows. Sect. II presents the background and related work, focusing on aggregate programming and its available tools. Sect. III details the motivation, architecture and characteristics of FCPP. Sect. IV compares FCPP with the other existing tools for simulation and execution of aggregate programs. Finally, Sect. V concludes with plans of future development.

II. Background and Related Work

The challenge of programming self-organising systems has been addressed in a variety of different contexts, each with its own goals and starting points. These various approaches can be grouped in few main clusters [1]: *(i)* methods abstracting away devices or networks (e.g., TOTA [18], MapReduce [19]); *(ii)* geometrical or topological pattern languages (e.g., self-healing geometries [20]); *(iii)* languages for information retrieval and routing (e.g., TinyLime [21]); *(iv)* general space-time computing models (e.g., StarLisp [22], aggregate programming [4]).

In this paper, we focus on the aggregate programming approach and Field Calculus language, in which a whole network runs a single program by periodically executing its main function asynchronously in all of its devices. Communication between devices is realised through low-level broadcasts, and modelled by the *neighbouring field* data type, which is essentially a map from neighbour device identifiers to relative values. Neighbouring fields are manipulated through "point-wise map" and "fold" operations, and created by specific constructs which automatically match outgoing and incoming messages—thanks to stack tracing techniques—in a way that ensures safe functional composition.

III. FCPP: Field Calculus in C++

A. Motivation and Application Scenarios

FCPP is C++14 library implementing the Field Calculus with tools for simulation of distributed systems, featuring *(i)* an *extensible* component-based software architecture, suitable to be customised for different application scenarios (e.g. IoT deployment, simulation, self-organising cloud) by addition of components targeting specific needed functionalities; *(ii)* a *performance-oriented* implementation based on compile-time optimisations, and designed to support parallel execution of a simulated system[5] or self-organising cloud application.

At the current state, the FCPP library only contains components supporting the simulation of distributed systems; and is already able to reduce significantly the simulation cost, speeding up the development process of new distributed algorithms.

Furthermore, the two features described above ensure the possibility of conveniently extending it to cover additional application scenarios, for which the previously available tools were ineffective. In particular *(i)* deployments on microcontroller-based systems, which do not usually meet the minimum performance requirements of existing implementations; and *(ii)* self-organising cloud applications, which require fine-grained parallelism in order to scale and for which performance improvements translate in a cost reduction.

The choice of C++ as programming language (without external dependencies such as e.g. Aspect C++), was driven by the need of targeting most existing platforms, including microcontrollers, for which a C++ compiler is usually available while it is not necessarily the case for other performance-oriented competing languages (e.g. Rust, Go).

B. Extensible Software Architecture

The FCPP library consists of 40 header files (as of the writing of this paper), divided in three main conceptual layers represented in Fig. 1: *(i)* C++ data structures of general use; *(ii)* components; *(iii)* aggregate functions. The first layer comprises data structures needed by the second layer either for their internal implementation or for the external specification of their options, but also data structures designed for the third layer of aggregate functions: an extension of C++ arrays and tuples supporting component-wise operations, a `vec<n>` class modelling physical vectors, and (most importantly) the `field<T>` class implementing the concept of *neighbouring field* (with point-wise map and fold operations).

The second layer defines the abstractions for *node* (single devices) and *net* (overall network orchestration, crucial in simulations and cloud-oriented applications). In an FCPP application, the two types `node` and `net` are obtained by combining a chosen sequence of components [23], each of them providing a needed functionality, in a *mixin*-like fashion [24], [25]. In order to provide compile-time mixin-like composition, components are templated classes with the following structure.

```
template <class... Ts>
struct my_functionality {
  template <class F, class P>
  struct component : public P {
    class node : public P::node { ... };
    class net : public P::net { ... };
  };
};
```

The `my_functionality` component above receives a variable list of options `Ts` as template parameters. Then, its `component` sub-class receives as further parameters both a *parent* component composition `P`, and the *final* composition of all components `F`.[6] Then, the `node` and `net` nested classes contain the additional functionalities provided by component `my_functionality`, as methods possibly depending on both parents' methods in `P` and/or final methods in `F`.

The dependencies between currently provided components are depicted in Fig. 1 (layer 2—components), and are enforced through `static_assert` checks. The number of dependencies has been kept as low as possible, and thanks to mixin composition, they don't prevent the substitution of a "required" component for another offering an analogous interface. Overall,

[5] We remark that parallel execution of a single simulated system is currently not supported by the other available Field Calculus simulators.

[6] In order to feed the final component composition to each of its contained components, an instance of the *Curiously Recursive Template Pattern* (CRTP, first introduced in [26]) has been used.

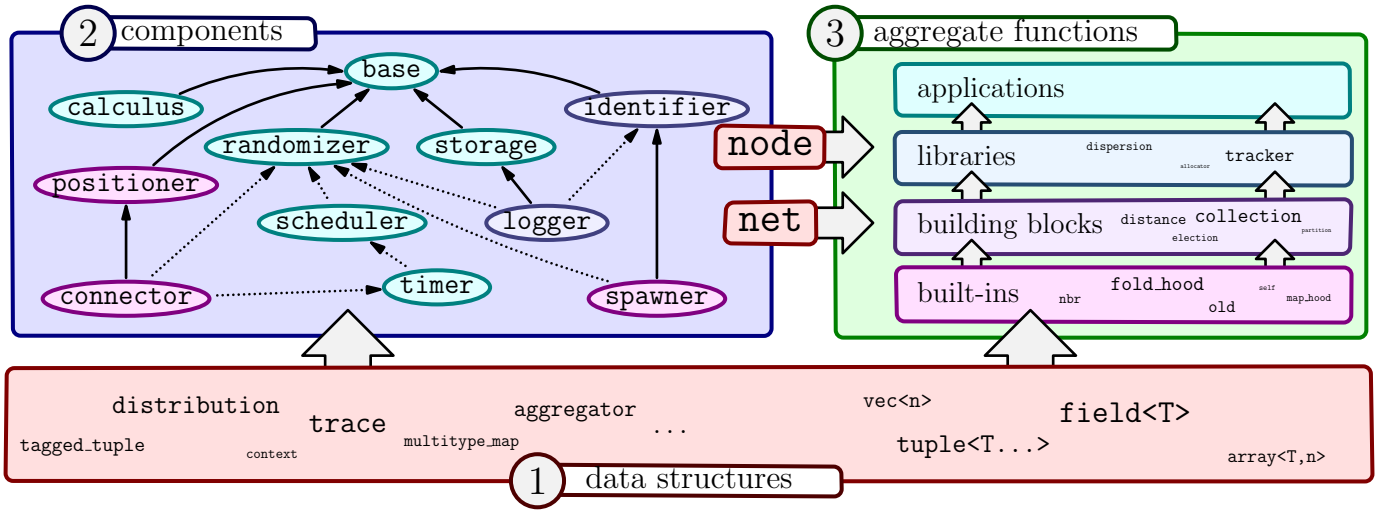

Fig. 1. Representation of the software architecture of FCPP as the combination of three main layers: *data structures* for both other layers, *components* which provide node and network abstractions to *aggregate functions*. Components are categorized as general purpose (cyan), pertaining to simulations or cloud-based applications (violet), and simulation specific (magenta). Dependencies between them can be either *hard* (solid), for which the pointed component is required as an ancestor of the other; or *soft* (dotted), for which the pointed component is not required, but if present, it should be an ancestor of the other component.

TABLE I
FCPP COMPONENTS

Component	Provides
`base`	basic update interface, node unique identifiers, reference to net objects in nodes
`calculus`	stack trace, context and exports, to be used indirectly through aggregate functions
`identifier`	manages a collection of node objects indexed by UID (creation, removal, access)
`logger`	periodically aggregates values from a `storage` component across nodes into a given stream
`physical_connector`	simulates periodic broadcasts of messages, determining whether connection is possible based on the simulated `physical_position`
`physical_position`	physical evolution of n-dimensional position, velocity, acceleration vectors with friction
`randomizer`	random number generator, functions producing random integer or floating point values
`scheduler`	automatically schedules rounds
`spawner`	automatically creates nodes in the `identifier` (possibly using random distributions)
`storage`	holds a collection of values in a tuple-like fashion
`timer`	maintains temporal information and allows interactive scheduling of rounds

the component system allows to reuse specific functionalities across different systems, changing only those needed. For example, the `physical_connector` component responsible of message exchanges in simulated systems, could be substituted for a `networked_connector` component with similar interface to be used for exchanging messages in deployed systems through a broadcast network interface. A comprehensive list of the components currently implemented in FCPP, together with a description of their role, is given in Table I.

The third layer provides the actual implementation of Field Calculus programs, as templated functions with a `node` parameter, and an identifier of the point where the function was called as a further parameter (needed for implementing the automatic alignment mechanism of Field Calculus for matching messages across nodes). As an example, the following function estimates distances from devices where `source` is true through the *Adaptive Bellman Ford* algorithm [27], [28].

```
template <class node_t>
double abf(node_t& node, trace_t call_point, bool source) {
    trace_call trace_caller(node.stack_trace, call_point);
    return nbr(node, 0, INF, [&] (field<double> d) {
        double v = source ? 0.0 : INF;
        return min_hood(node, 1, d + node.nbr_dist(), v);
    });
}
```

Notice that a reference to the `node` object is explicitly provided, allowing to access the functionalities provided by nodes: in this case, `node.nbr_dist()` which is a built-in function returning a `field<double>` of estimated distances with neighbour nodes. The `call_point` parameter is used to update the `node.stack_trace` (first line, which is in common to all aggregate functions), and is provided to aggregate function calls (such as `nbr` and `min_hood` above) as an increasing index. The need for an explicit managing of the stack trace can be circumvented by resorting to C++ macros; through which the above function can be rewritten as in the following.

```
DEF() double abf(ARGS, bool source) { CODE
    return nbr(CALL, INF, [&] (field<double> d) {
        double v = source ? 0.0 : INF;
        return min_hood(CALL, d + node.nbr_dist(), v);
    });
}
```

This function repeatedly updates a distance estimate, starting from estimates of neighbour devices gathered in `field<double> d`, and setting the current estimate to either 0 if `source` is true, or to the minimum sum of a neighbour estimate with the estimated distance to that neighbour otherwise. Notice that according to the aggregate programming paradigm, aggregate functions are themselves organised in stacked lay-

aggregate function declaration

F ::= DEF($T*$) t d(ARGS, t x*) {CODE return e; }

aggregate expression

e ::= x | ℓ | t{e*} | ue | e o e | p(e*) | node.c(e*) | f(CALL, e*)
| t x = e; e | [&](t x*)->t {return e; } | e ? e : e

type	aggregate function
t ::= T | bt | tt<$t*, \ell *$ >	f ::= b | d

built-in aggregate functions

b ::= self | mod_self | map_hood | fold_hood | old | nbr

Fig. 2. Syntax of FCPP aggregate functions.

ers of increasing abstraction and complexity: starting from built-in functions without a definition as aggregate functions (c.f. Sect. III-C); through building blocks of common use (e.g. leader election, distance estimation, data summarisation, c.f. Sect. IV-C), available as library functions in FCPP; to domain-specific library functions and final applications.

C. Programming Language

A restrictive syntax for aggregate functions is given in Fig. 2, reducing the expressive possibilities of FCPP to a minimal set of constructs matching the ones in Field Calculus. Following regular expression notation, we use $*$ to indicate an element that may be repeated multiple times (possibly zero), assuming that repetitions are always comma-separated.

An *aggregate function declaration* consists of a declaration of (any number of) type variables DEF($T*$), followed by the return type t and the function name d, followed by a parenthesized sequence of comma-separated arguments t x (prepended by the keyword ARGS), followed by an *aggregate expression* e (within brackets and keywords CODE return).

Notice that the definition of types follows the syntax of C++: a type t is either a type variable T, a non-templated type bt (such as bool, int, double), or a templated type tt (such as field, array, tuple) with some type and literal arguments provided tt<$t*, \ell *$ > (e.g. field<double>, array<int, 3> or tuple<vec<2>, bool, field<double>>).

Aggregate expressions can be either:

- a *variable* identifier x (either a function argument or defined in a let-style statement—see below);
- a *literal value* ℓ expressible in C++ (e.g. an integer or floating-point number, true, false, string literals, etc.);
- an *object* of type t built through a class constructor call t{e*} with arguments e;
- an *unary operator* u (e.g. $-$, $\sim$, !, etc.) applied to e;
- a *binary operator* e o e (e.g. $+$, $*$, and, or, etc.);
- a *pure function call* p(e*), where p is a basic C++ function which does not depend on node information nor message exchanges (or an anonymous aggregate function—see below);[7]
- a *component function call* node.c(e*), where c is a function provided by some component (see Sect. III-B), depending on node information but not on messages;
- an *aggregate function call* f(CALL, e*), where f can be either a defined aggregate function name d or an aggregate built-in function b (corresponding to either field manipulation or Field Calculus constructs—see below);
- a *let-style statement* t x = $\mathtt{e}_1$; $\mathtt{e}_2$, declaring a variable x of type t with value $\mathtt{e}_1$ referable in $\mathtt{e}_2$;
- an *anonymous function* [&](t x*)->t_r {return e; } with arguments t x, return type t_r, and body e: since anonymous functions capture variables defined in their scope, they also capture the node variable which allows their body to be an aggregate expression, even though they are applied as pure functions;
- a *conditional branching* expression $\mathtt{e}_{\text{guard}}$? $\mathtt{e}_\top$: $\mathtt{e}_\bot$, such that $\mathtt{e}_\top$ is evaluated and returned if $\mathtt{e}_{\text{guard}}$ evaluates to true, while $\mathtt{e}_\bot$ is evaluated and returned if $\mathtt{e}_{\text{guard}}$ evaluates to false.

Comparing the syntax in Fig. 2 to the one of the Field Calculus [12], [13], [29], it is remarkably similar in structure, although with some organisational differences. In particular, the coordination constructs nbr, rep, share (and variants) that are primitive concepts in Field Calculus are here modelled through built-in aggregate functions old and nbr. These two built-in functions are overloaded to several different signatures:

- old(CALL, v_0, v) with v_0, v of type t returns the value fed as second argument v in the *previous round* of computation (thus introducing one round of delay), defaulting to v_0 if no such value is available;
- old(CALL, v) is a shorthand for old(CALL, v, v);
- old(CALL, v_0, f) corresponds to the rep operator of Field Calculus, and computes the result of applying f (which can either be an anonymous aggregate function or a pure function) of signature (t)->t to the value of the whole old function at the previous computation cycle (using v_0 if no such value is available);
- nbr(CALL, v_0, v) with v_0, v of type t returns the *neighbouring field* of values fed as second argument v in the previous round of computation of *neighbour nodes*, defaulting to v_0 for the current node if no such value is available for it;
- nbr(CALL, v) is a shorthand for nbr(CALL, v, v), and corresponds to the nbr operator of Field Calculus;
- nbr(CALL, v_0, f) corresponds to the share operator of Field Calculus [29], and computes the result of applying f (which can either be an anonymous aggregate function or a pure function) of signature (field<t>)->t to the neighbouring field of values of the whole nbr function at the previous computation cycle of neighbour nodes (using v_0 for the current node if no such previous value is available for it).

The other built-in aggregate functions currently available are:

- self(CALL, ϕ), which given a value ϕ of field<t> type returns the value $\phi(i)$ taken by the neighbouring field ϕ

[7] Unary and binary operators also qualify as pure functions: they only differ by the usage of the infix notation for calls.

for the current node (of identifier $i =$ `node.uid`);

- `mod_self`(`CALL`, ϕ, v), which given a value ϕ of `field`<t> type, returns the same value with $\phi(i)$ changed to v, where $i =$ `node.uid`;
- `map_hood`(`CALL`, $f, v*$) which applies f point-wise to a sequence of local or field values $v*$;
- `fold_hood`(`CALL`, f, ϕ) which *folds* the values in the range of ϕ of `field`<t> type through the commutative and associative binary operator f of type (t, t)->t, reducing them to a single value of type t;[8]
- `fold_hood`(`CALL`, f, ϕ, v) which folds ϕ as above, using v instead of the value of ϕ for the current device: in other words, it is equivalent to `fold_hood`(`CALL`, f, `mod_self`(`CALL`, ϕ, v).

IV. Tool Comparison

A. Alchemist and the Simulation Features of FCPP

Alchemist [15] is a general-purpose, flexible simulator of distributed systems with an extensive range of features. It can simulate bidimensional environments, indoor environments with obstacles (described as black and white images), and real-world maps (with support for GPS traces and navigation on roads). It is integrated with four "incarnations", i.e., languages specifying distributed behaviour: *protelis* and *scafi* (languages for aggregate programming), *sapere* (tuple-based language for distributed computing), *biochemistry* (language describing chemical-like reactions). Batches of simulations can be run in grid computing systems through Apache ignite.

Although FCPP is designed as a flexible and easily extensible platform, the currently supported range of features is more limited: it lacks several functionalities present in Alchemist, although while also providing few features that Alchemist lacks. It can be easily extended to deployed systems, and it can simulate bidimensional and *tridimensional* environments, with a physically accurate model based on position, velocity, acceleration and friction (features not currently available in Alchemist). On the other hand, it doesn't yet have support for obstacles, real-world maps and GPS traces. Furthermore, it only supports one language for specifying distributed behaviour (discussed in Sect. IV-B), following the aggregate programming paradigm.

Besides the difference in features and applications supported, the two tools also diverge in their overall structural approach. Alchemist (like e.g. Repast and NetLogo) is a *stand-alone application* running on the JVM, which can be used to run experiments. On the other hand, FCPP (like e.g. ns-3) is a *library*, used to develop experiments as applications importing it. The two different approaches imply corresponding trade-offs: *(i)* in stand-alone simulators, multiple runnable experiments share the application, which can be downloaded and built only once; while in simulation libraries every runnable experiment need to include and be compiled with (part of) the library, forcing parts of the simulation library to be duplicated across the different experiments; *(ii)* simulation libraries can be more performant due to compile-time optimizations, and produce small runnable experiments where only the parts of the library needed are included; while stand-alone simulators have generally an higher performance and disk space overhead.

B. Protelis, Scafi and the Language Features of FCPP

The FCPP language for aggregate programming implements the abstract Field Calculus language, as Protelis and Scafi do. Thus, the basic features and expressive power of the three languages mostly coincide, with few notable differences. In fact, the main differences among them boil down to Scafi and FCPP being *internal* domain specific languages (DSL), implemented directly in Scala and C++ respectively, while Protelis is an *external* DSL interpreted in the JVM.

Thus, the Protelis syntax is more neat and specifically designed for field computations, while the syntaxes of Scafi and FCPP can only partially mimic the abstraction level of Field Calculus, being tied by the specific syntactic restrictions of their host languages (which are stronger for C++ than Scala, making FCPP the least "clean" language out of the three). On the other hand, the more complex syntaxes of Scafi and FCPP may turn out to be simpler to familiarise with for programmers already proficient in Scala and C++ (the latter being particularly well-known among programmers).

On the other hand, internal DSL are more performant and can easily inherit most of the features of their host languages. Among those, an expressive type system: the most expressive type system is available in Scafi (inheriting types from Scala), followed by FCPP (expressive on parametric types, but less suited for functional types), and finally by Protelis which has no type system available at the moment. Other constructs inherited by the host language may also prove useful in some circumstances: for example, C++ cycles are available in FCPP (both natively within pure C++ functions, and in aggregate functions through a custom cycle alignment support).

Finally, Scafi was designed as a customised version of Field Calculus based on `fold_hood` operations, without explicit `field` types. In order to obtain the same behaviour of Field Calculus, a Scafi programmer needs to manually avoid certain function signatures (e.g. arguments of `field` type), thing that may require some care and syntactic tweaks.[9] On the other hand, Protelis and FCPP are natively designed on the Field Calculus with neighbouring fields.

C. Case Study

In order to measure the performance improvement of FCPP for the simulation of aggregate systems with respect to existing Field Calculus implementations and simulators, we translated the recent simulation scenario in Audrito et al. [11] into FCPP, then compared the CPU time, RAM memory and hard disk occupancy needed to *run* the simulations. We focused on *running* instead of *building*, since the running requirements determine which possible architectures can support the execution of these

[8] In Field Calculus, a neighbouring field always has at least a value for the current node; thus, folding is well-defined.

[9] True `field` types are currently available in Scafi as an additional library; however, they need to be manually aligned.

TABLE II
PERFORMANCE COMPARISON

	HDD	RAM avg/peak	CPU
FCPP	0.4 MB	0.06 / 0.07 GB	0:04:32
(1.5GB)		1.96 / 2.08 GB	8:03:10
(2.0GB)		2.48 / 2.56 GB	6:39:34
(2.5GB)	174 MB .jar	2.88 / 3.05 GB	5:45:19
Alchemist+Protelis (3.0GB)	626 MB Gradle	3.33 / 3.55 GB	5:13:49
(3.5GB)	*+100MB (JVM)*	3.81 / 4.09 GB	4:45:06
(4.0GB)		4.10 / 4.45 GB	4:29:47
(4.5GB)		4.38 / 4.77 GB	4:17:45

implementations, and also since the *building* phases of standalone simulators (Alchemist) and simulator libraries (FCPP) are very different in nature and purpose.

1) Scenario Description: The benchmark simulation scenario chosen [11] focuses on the problem of *data aggregation*, where a set of values v_i, each locally available in a device i of the network, has to be collapsed into a single aggregate value $\bigoplus_i v_i$ (e.g. the sum, mean, maximum, minimum, etc.), available in a single selected device (a *leader*) of the network. The aggregation routine takes as input a set of distance estimates d_i from the leader to guide the information flow, so that partial aggregates in a device i are computed by combining partial aggregates of (some) neighbour devices j with larger distances $d_j > d_i$.

This case study compares the effectiveness of several distributed data aggregation routines (single-path [6], multi-path [6], weighted multi-path [11]), each combined with different distance estimation algorithms guiding the aggregation (BIS [8], FLEX [10], ULT [9]). For each of these combinations of algorithms, two aggregation scenarios are considered: *(i) device counting*, where the leader device has to compute the total number of devices in the network; *(ii) progress tracking*, where the leader has to compute the maximum progress reached by a device in the network.

In both cases, the simulation comprises 1000 simulated devices with a $100m$ connection radius walking through random waypoints in a $2000m \times 200m$ rectangular area. Each device performs rounds of computation with an average frequency of 1Hz, for a total $500s$ of simulated time.

2) Performance Results: The tests were run on a MacBook Pro (13-inch, Mid 2012), with a 2.9 GHz Intel Core i7 dual-core CPU, and 8 GB of DDR3 RAM at 1600 MHz. We compared the implementation in Audrito et al. [11] (based on Alchemist [15] and Protelis [16]) for several heap size settings, with an equivalent implementation in FCPP.[10] We averaged the RAM and CPU usage across 80 runs for FCPP and 80 runs for Alchemist (the latter split into 8 different heap size settings), with a relative standard error between runs always below 7% (3.5% in average). The resulting performance statistics for the various implementations are summarised in Table II.

[10]The implementation is available at https://github.com/fcpp/sample-project as the sample project of FCPP; and for the Alchemist and Protelis implementation at https://bitbucket.org/Harniver/aamas19-summarising/src/fcpprun.

The negligible hard disk space (HDD) necessary to run the FCPP program (440 KB) correspond to the compiled executable size, without any other additional external requirements.[11] For Alchemist and Protelis, the execution of a program requires the presence of the JVM (about 100 MB) and of the whole simulator and language, due to the different structural approach (see Sect. IV-A). Executing the simulation with Gradle,[12] which automatically handles building and dependencies, occupies a total 626 MB of disk space. By manually managing build and dependencies, it is possible to reduce the space needed to 174 MB (size of the redistributable *jar* file for the Alchemist 9.3.0 release), JVM excluded.

For Alchemist and Protelis, the amount of RAM to be used is manually set as an option to the JVM. If the amount chosen is too low, the execution will crash with an out-of-memory error: in the test at hand, this happened consistently whenever the maximum heap size of the JVM was set to 1.0 GB. If the amount chosen is large enough to allow execution, but too close to the lower limit, the execution slows down significantly due to an excessive garbage collection load. If the amount chosen is yet larger, execution speed stops reducing significantly, while memory occupancy keeps increasing, filling up all the available space. Thus, tuning the "maximum heap size" parameter is a crucial step for an efficient execution of an Alchemist and Protelis simulation.

We tested maximum heap sizes between 1.0 and 4.5 GB with steps of 0.5 GB. With 1.0 GB the simulation crashed consistently, thus no results are reported in the table. For heap sizes up to 3.5 GB, the actual RAM used peaked at about that maximum heap size plus an extra 600 MB, with an average usage at about extra 300-500 MB. For heap sizes of 4.0 and 4.5 GB, the RAM used peaked at about an extra 300-450 MB with an average close to the heap size. On the other hand, the amount of RAM memory used in FCPP is fixed, depending only on the amount of space actually needed, and peaked at 68 MB which is $30\times$ to $70\times$ lower than the RAM needed by Alchemist and Protelis.

Regarding the total CPU time used for the execution, the FCPP implementation completed after 4 minutes and 32 seconds in average, while Alchemist and Protelis needed 4 to 8 hours depending on the heap sizes ($60\times$ to $100\times$ as much). Overall, the performance improvement of FCPP with respect to Alchemist and Protelis is significant; reducing by orders of magnitude the cost of simulating a distributed system, and paving the way towards the execution of aggregate programs on low-end devices, and cost-effective self-organising cloud applications.

V. CONCLUSIONS AND FUTURE DEVELOPMENT

We presented FCPP, a novel implementation of the Field Calculus language as a C++14 library, together with tools for simulation and execution of distributed systems. Although its

[11]For the interested reader, building the FCPP experiment required 39.4 seconds of CPU time and occupied 940MB of space on disk. However, once built, only the executable is needed for running the experiments.

[12]Build tool available at https://gradle.org.

scope is currently narrower than that of existing aggregate programming tools (Alchemist, Protelis, Scafi), it has potential for extensibility due to its component-based architecture, and it already covers some features and application scenarios that are not adequately covered by the other tools. Furthermore, it grants measurable efficiency improvements, which we quantified by re-implementing a recent case study from literature.

In future work, we plan to extend the comparison with the state-of-the-art by re-implementing the case study also in Scafi (simulated with Alchemist or with the built-in Scafi simulator), and measure performance more accurately with a profiler, in order to separate the resources needed for the *network simulation* and *language execution* aspects of the computation.

Since the FCPP library is still in its early stage of life, far-reaching development plans are in order: developing new components for simulation, self-organising cloud scenarios and microcontrollers; adding support for interactive simulations by importing the C++ graphical user interface of Proto [14]; extending the library of aggregate functions; improving documentation and support; evaluating the usability of the library. In particular, we plan to perform a case study on a self-organising cloud scenario in the near future, and on a deployment of microcontrollers in a mid-term future. Furthermore, a tool for translating a Kotlin-like (and Protelis-like) syntax into FCPP aggregate functions is already in its early stage of development, with the purpose of offering a cleaner syntax interface while retaining the benefits of the library.

Acknowledgment

We thank Jacob Beal, Ferruccio Damiani, Mirko Viroli for the fruitful discussions on aggregate computing and for the precious suggestions without which this work would not have been possible. Special thanks go to Danilo Pianini and Roberto Casadei for sharing their experience on programming and simulating aggregate systems, which constituted the grounds upon which this work has been built.

References

[1] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, "Organizing the aggregate: Languages for spatial computing," in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, 2013, pp. 436–501.

[2] S. Dobson, S. G. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 1, no. 2, pp. 223–259, 2006.

[3] M. Mamei, R. Menezes, R. Tolksdorf, and F. Zambonelli, "Case studies for self-organization in computer science," *Journal of Systems Architecture*, vol. 52, no. 8-9, pp. 443–460, 2006.

[4] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the internet of things," *IEEE Computer*, vol. 48, no. 9, pp. 22–30, 2015.

[5] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, "From distributed coordination to field calculus and aggregate computing," *Journal of Logical and Algebraic Methods in Programming*, vol. 109, 2019.

[6] M. Viroli, G. Audrito, J. Beal, F. Damiani, and D. Pianini, "Engineering resilient collective adaptive systems by self-stabilisation," *ACM Transactions on Modeling and Computer Simulation*, vol. 28, no. 2, pp. 16:1–16:28, 2018.

[7] J. Beal, M. Viroli, D. Pianini, and F. Damiani, "Self-adaptation to device distribution in the internet of things," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 12, no. 3, pp. 12:1–12:29, 2017.

[8] G. Audrito, F. Damiani, and M. Viroli, "Optimal single-path information propagation in gradient-based algorithms," *Science of Computer Programming*, vol. 166, pp. 146–166, 2018.

[9] G. Audrito, R. Casadei, F. Damiani, and M. Viroli, "Compositional blocks for optimal self-healing gradients," in *11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE Computer Society, 2017, pp. 91–100.

[10] J. Beal, "Flexible self-healing gradients," in *ACM Symposium on Applied Computing (SAC)*. ACM, 2009, pp. 1197–1201.

[11] G. Audrito, S. Bergamini, F. Damiani, and M. Viroli, "Effective collective summarisation of distributed data in mobile multi-agent systems," in *18th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. International Foundation for Autonomous Agents and Multiagent Systems, 2019, pp. 1618–1626.

[12] G. Audrito, M. Viroli, F. Damiani, D. Pianini, and J. Beal, "A higher-order calculus of computational fields," *ACM ACM Transactions on Computational Logic*, vol. 20, no. 1, pp. 5:1–5:55, 2019.

[13] G. Audrito, J. Beal, F. Damiani, and M. Viroli, "Space-time universality of field calculus," in *20th International Conference on Coordination Models and Languages (COORDINATION)*, ser. Lecture Notes in Computer Science, vol. 10852. Springer, 2018, pp. 1–20.

[14] J. Beal and J. Bachrach, "Infrastructure for engineered emergence on sensor/actuator networks," *IEEE Intelligent Systems*, vol. 21, no. 2, pp. 10–19, 2006.

[15] D. Pianini, S. Montagna, and M. Viroli, "Chemical-oriented simulation of computational systems with ALCHEMIST," *Journal of Simulation*, vol. 7, no. 3, pp. 202–215, 2013.

[16] D. Pianini, M. Viroli, and J. Beal, "Protelis: practical aggregate programming," in *30th ACM Symposium on Applied Computing (SAC)*. ACM, 2015, pp. 1846–1853.

[17] M. Viroli, R. Casadei, and D. Pianini, "Simulating large-scale aggregate mass with alchemist and scala," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*, ser. Annals of Computer Science and Information Systems, vol. 8. IEEE, 2016, pp. 1495–1504.

[18] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: The TOTA approach," *ACM Transactions on Software Engineering Methodologies*, vol. 18, no. 4, pp. 15:1–15:56, 2009.

[19] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[20] A. Kondacs, "Biologically-inspired self-assembly of two-dimensional shapes using global-to-local compilation," in *18th International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, 2003, pp. 633–638.

[21] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco, "Mobile data collection in sensor networks: The TinyLime," *Pervasive and Mobile Computing*, vol. 1, no. 4, pp. 446–469, 2005.

[22] C. Lasser, J. Massar, J. Miney, and L. Dayton, *Starlisp Reference Manual*. Thinking Machines Corporation, 1988.

[23] M. D. McIlroy, J. Buxton, P. Naur, and B. Randell, "Mass-produced software components," in *1st international conference on software engineering*, 1968, pp. 88–98.

[24] H. I. Cannon, "Flavors: A non-hierarchical approach to object-oriented programming," Artificial Intelligence Laboratory, MIT, USA, Tech. Rep., 1979.

[25] G. Bracha and W. R. Cook, "Mixin-based inheritance," in *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) / European Conference on Object-Oriented Programming (ECOOP)*. ACM, 1990, pp. 303–311.

[26] P. S. Canning, W. R. Cook, W. L. Hill, W. G. Olthoff, and J. C. Mitchell, "F-bounded polymorphism for object-oriented programming," in *4th International Conference on Functional Programming Languages and Computer Architecture (FPCA)*. ACM, 1989, pp. 273–280.

[27] N. Chandrachoodan, S. S. Bhattacharyya, and K. J. R. Liu, "Adaptive negative cycle detection in dynamic graphs," in *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2001, pp. 163–166.

[28] G. Audrito, F. Damiani, M. Viroli, and E. Bini, "Distributed real-time shortest-paths computations with the field calculus," in *Real-Time Systems Symposium, (RTSS)*. IEEE Computer Society, 2018, pp. 23–34.

[29] G. Audrito, J. Beal, F. Damiani, D. Pianini, and M. Viroli, "The share operator for field-based coordination," in *21th International Conference on Coordination Models and Languages (COORDINATION)*, ser. Lecture Notes in Computer Science, vol. 11533. Springer, 2019, pp. 54–71.