Kompics Scala: Narrowing the Gap between Algorithmic Specification and Executable Code (Short Paper)

Lars Kroll
KTH Royal Institute of Technology
Stockholm, Sweden
lkroll@kth.se

Paris Carbone
KTH Royal Institute of Technology
Stockholm, Sweden
parisc@kth.se

Seif Haridi
KTH Royal Institute of Technology
Stockholm, Sweden
haridi@kth.se

Abstract

Message-based programming frameworks facilitate the development and execution of core distributed computing algorithms today. Their twofold aim is to expose a programming model that minimises logical errors incurred during translation from an algorithmic specification to executable program, and also to provide an efficient runtime for event pattern-matching and scheduling of distributed components. Kompics Scala is a framework that allows for a direct, streamlined translation from a formal algorithm specification to practical code by reducing the cognitive gap between the two representations. Furthermore, its runtime decouples event pattern-matching and component execution logic yielding clean, thoroughly expected behaviours. Our evaluation shows low and constant performance overhead of Kompics Scala compared to similar frameworks that otherwise fail to offer the same level of model clarity.

CCS Concepts • **Software and its engineering → Domain specific languages**; • **Computing methodologies** → *Distributed programming languages*; Concurrent programming languages;

Keywords component model, message-passing, distributed systems architecture

ACM Reference Format:
Lars Kroll, Paris Carbone, and Seif Haridi. 2017. Kompics Scala: Narrowing the Gap between Algorithmic Specification and Executable Code (Short Paper). In *Proceedings of 8th ACM SIGPLAN International Scala Symposium (SCALA'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3136000.3136009

SCALA'17, October 22–23, 2017, Vancouver, Canada

1 Introduction

At the heart of the now ubiquitous distributed systems sits a number of core distributed algorithms with associated correctness proofs for their formal description, as seen for example in common textbooks like Cachin et al. [2011]. Message-based programming frameworks (e.g., Akka [Akk 2009]) are widely used today in order to express such formal algorithms in code and therefore allow their correct execution on distributed deployments. A well known problem that often leads to critical errors is that the correctness of an algorithmic specification does not trivially translate into the correctness of any particular implementation, as some interpretation and adaptation is typically necessary to convert a formal description into a practical implementation. Usually, these bugs can easily be traced using various methods of unit testing, but the area of distributed systems tends to be the exception to this rule, as testing for the correctness of distributed properties is often very challenging and error prone in itself. It is thus reasonable to attempt to avoid the introduction of such "transcription" errors as best as possible, by allowing the actual code to resemble the formal description closely. This has been one of the goals of the Kompics framework [Kom 2009; Arad et al. 2012] at its inception which has therefore been used thoroughly in distributed systems education, in addition to system implementations (e.g., CaracalDB [Car 2013], GVoD [GVo 2014]).

However, the initial choice of Java as host language for Kompics' DSL has shown itself to be problematic, primarily in its verbosity, but also particularly due to the mismatch between the pattern matching formalism from the textbooks and the simple type-based match of the implementation as shown in figure 1.

This was the driving need behind a major redesign of the Kompics' programming and execution model based in Scala, that we present here, which allows for a more direct, intuitive mapping of a formal algorithm description to code execution. Our approach exploits Scala's pattern matching capabilities at its fullest and is partly inspired by Akka's Scala DSL. The implementation allows seamless interoperability of Kompics components written in Java and Scala and reuses much of the underlying framework code from the Java implementation. Furthermore, we introduce a fundamental decoupling

```scala
package se.sics.kompics;

case class Broadcast(m: Any) implements KompicsEvent

object BestEffortBroadcast extends sl.Port {
    request[Broadcast];
    indication[Deliver];
}
class BasicBroadcast(init: Init) extends sl.ComponentDefinition {
    val beb = provides[BestEffortBroadcast];
    val pl = requires[PerfectPointToPointLink];

    private val Init(pi: Set[Address]) = init;

    beb uponEvent {
        case Broadcast(m) => handle {
            pi.foreach { q =>
                trigger (Send(q, m) -> pl);
            }
        }
    }

    pl uponEvent {
        case Deliver(p, m) => handle {
            trigger (Deliver(p, m) -> beb);
        }
    }
}
```

(a)

Algorithm 1 Basic Broadcast

Implements:

BestEffortBroadcast, **instance** *beb*.

Uses:

PerfectPointToPointLinks, **instance** *pp2p*.

1: **upon event** $\langle\ beb, Broadcast \mid m\ \rangle$ **do**
2: **for all** $q \in \Pi$ **do**
3: **trigger** $\langle\ pp2p, Send \mid q, m\ \rangle$

4: **upon event** $\langle\ pp2p, Deliver \mid p, m\ \rangle$ **do**
5: **trigger** $\langle\ beb, Deliver \mid p, m\ \rangle$

(b)

```java
package se.sics.kompics;

public class Broadcast implements KompicsEvent {
    public final Object m;
    public Broadcast(Object m) {
        this.m = m;
    }
}
public class BebDeliver implements KompicsEvent {
    public final Address p;
    public final Object m;
    public Deliver(Address p, Object m) {
        this.p = p;
        this.m = m;
    }
}
public class BestEffortBroadcast extends PortType {{
    request(Broadcast.class);
    indication(BebDeliver.class);
}}
public class BasicBroadcast extends ComponentDefinition {
    Negative<BestEffortBroadcast> beb = provides(BestEffortBroadcast.class);
    Positive<PerfectPointToPointLink> pl = requires(PerfectPointToPointLink.class);

    private final Set<Address> pi;

    public BasicBroadcast(Init init) {
        this.pi = init.pi;
        subscribe(broadcastHandler, beb);
        subscribe(deliverHandler, pl);
    }

    Handler<Broadcast> broadcastHandler = new Handler<Broadcast>() {
        public void handle(Broadcast event) {
            for (Address q : pi) {
                trigger(new Send(q, event.m), pl);
            }
        }
    };
    Handler<Deliver> deliverHandler = new Handler<Deliver>() {
        public void handle(Deliver event) {
            trigger(new BebDeliver(event.p, event.m), beb);
        }
    };

    public static class Init implements se.sics.kompics.Init<BasicBroadcast> {
        public final Set<Address> pi;
        public Init(Set<Address> pi) {
            this.pi = pi;
        }
    }
}
```

(c)

Figure 1. Basic Broadcast in Kompics Scala (a), textbook algorithm adapted from [Cachin et al. 2011] (b), and Kompics Java (c).

of pattern matching and message handler execution to integrate with the Kompics model of "match-then-schedule" execution, described in sections 2 and 4.

The rest of the paper is structured as follows: Section 2 introduces the base Kompics model and its Java-based implementation. In sections 3 and 4 we describe the new Kompics Scala DSL and its new event handling runtime mechanism respectively. We evaluate the impact of the solution, both in terms of performance and experiences in the context of education in section 5, and summarise our work in section 6.

2 The Kompics Component Model

In this section we give a short introduction to the Kompics component system [Arad et al. 2012; Arad 2013]. The core philosophy of the model is a strict decoupling of a service specification, called a *port*, and its implementation, called a *component*, which has no information about its environment including any other components providing services to it or making use of its services. This approach allows for a powerful hierarchical composition of large services from smaller building blocks and high modularity upon selecting particular implementations of required services upon deployment.

Semantics: Kompics is a programming model for distributed systems that implements protocols as event-driven *components* connected via *channels*. *Ports* provide a form of type system for events. They are declared as either required or provided by each component and define which event-types travel along the channels that connect them and in which direction. On a port type, the "service specification" for a port, events are declared as either *indications* or *requests*. Within a component that *provides* a port P with indication event I and request event R, only instances of I (and its subtypes) may be *triggered* ("sent") and only instances of R (or their subtypes) can be *handled* (see below). Conversely, within a component that *requires* P only instances of R can be *triggered* and only instances of I can be *handled*. The channels connecting ports provide first-in-first-out (FIFO) order exactly-once (per receiver) delivery and events are queued up at the receiving ports until the component is scheduled to execute them.

Component Scheduling: Kompics components are only scheduled if there are outstanding events queued on one or more of their ports. A component is guaranteed to be scheduled and executed by a single thread at a time and thus has exclusive access to its internal state without the need for further synchronisation. Different components, however, are scheduled in parallel in order to exploit the parallelism expressed in a message-passing program. When a component is scheduled it handles one event at a time, until either there are no more events queued at its ports or a configurable maximum number η_{max} of events to be handled is reached. After the component has finished handling events, it is placed at the end of the FIFO queue of components waiting to be scheduled, if it still has outstanding events. Tuning η_{max} enables developers to tradeoff increased throughput, where higher values maximise cache reuse through fewer component context switches, against fairness, that is avoiding starvation of components with fewer queued events.

Handler Execution: In contrast to Actor systems like Akka or Erlang [Armstrong 2003], events in Kompics are not addressed to components in any way, but are instead published on all connected channels. In this way the same event can be received by many components. The components themselves decide which events to handle and which to ignore by *subscribing* event *handlers* on their declared ports. Note that ignored messages are silently dropped, which is necessitated by the channel broadcasting model, that is to say,

```
case class Send(to: Address, payload: Any) extends KompicsEvent
case object Sent extends KompicsEvent
object Net extends sl.Port {
  request[Send];
  indication(Sent);
}
```

Listing 1. Kompics Scala Port Example.

as opposed to Erlang and Akka, in Kompics it is often completely correct to simply ignore a large number of events arriving at a particular port. In order to avoid unnecessary load on the scheduler, a Kompics component is only scheduled *after* checking that at least one subscription exists for the incoming event.

Matching of an event e of type E to a handler h with event-type E_h is based on the subtyping relationship <:, such that h matches e iff $E <: E_h$.

Distributed Execution: While Kompics is meant as a framework for programming distributed systems, everything in this paper only describes purely local execution within a single JVM. Kompics is philosophically opposed to any attempt at location transparency and thus networking in Kompics is not part of the core model, but instead a possible implementation of a component providing network transport is provided in the standard library, and any user may write their own to fulfil their particular requirements.

3 Kompics Scala DSL

Kompics Scala is not a complete re-implementation of the Kompics component model, but rather an extension of the Java library, reusing functionality wherever possible. In fact, the two libraries reside in separate name spaces to allow interoperability between Java and Scala (`se.sics.kompics`[1] and `se.sics.kompics.sl` respectively). This section will describe the "cosmetic" differences in Kompics Scala, while section 4 will deal with implementation differences. A direct comparison between Kompics Scala, Java, and textbook pseudocode of a simple algorithm (basic broadcast) can be seen in figure 1.

Ports and Events: An event may be any **class** or **object** that extends the Java `KompicsEvent` trait (interface). In order to make proper use of Scala's pattern matching most events should be a **case class** or a **case object**.

Any port type should now properly be an **object** extending `sl.Port`, as port types were always supposed to be singletons. Both `request` and `indication` events may be given as type tag or as object to their respective methods, as seen in listing 1.

Components and Handlers Kompics Scala components extend `sl.ComponentDefinition` instead of `ComponentDefinition`, which provides type tag and object parameter versions of the Java methods for requiring/providing ports, as seen in listing 2. Handler creation and subscription now happens in a single step via `<port> uponEvent { ... }` to avoid the bug of neglecting

[1]For brevity, we will omit the `se.sics.kompics` prefix in the rest of the paper.

```
class ExampleC extends sl.ComponentDefinition {
  val net = requires(Net);
  val timer = requires[Timer];
  ctrl uponEvent {
    case _: Start => handle {
      trigger (Send(dst, "message") -> net);
    }
  }
  net uponEvent {
    case Sent => handle {
      println("Message was sent!");
    }
  }
}
```

Listing 2. Kompics Scala Component Example.

```
class ParentC extends sl.ComponentDefinition {
  val timerC = create[JavaTimer];
  val exampleC = create[ExampleC];
  connect[Timer](timerC -> exampleC);
}
```

Listing 3. Kompics Scala Channel Example.

to subscribe a handler to a port, which in our experience is very common. The inside of a handler block is a partial function mapping events to functions (closures) that handle them, acquired via the `handle` syntactic sugar for creating a function `() => Unit`. Section 4 explains the reason for using this particular approach to event matching, as opposed to writing handler code directly on the right-hand side of a case-statement.

Kompics Scala also allows `trigger` to be used with a tupled argument instead of two parameters, which, coupled with the implicit conversion from `(event -> port)`, leads to more readable code for triggering `event` on `port`.

Channels: A similar notation is used in `connect` statements, with the arrow here going from (service) provider to consumer ("requirer"), as shown in listing 3. The use of a type tag here avoids the repetition of the port type as in Java, e.g., `connect(timerC.provided(Timer.class), exampleC.required(Timer.class))`.

4 Delayed Handler Execution

As described in section 2, scheduling of components in Kompics happens only after a matching subscription is found on the destination port. Event broadcasting and initial matching happens on the thread executing the component triggering the event, i.e. *sending thread*. That makes port queues act as a synchronisation point between components (and the external world). In Kompics Java matching happens via a sequence subtype checks. As soon as any matching handler is found for a component, the event gets queued on the port and that component is scheduled for execution without attempting to match any further handlers in the same component. Once the component is executed on a thread, termed the *receiving thread*, it begins to match handlers with its queued events until all matches are found and then executes all handlers in the order they were matched (no other ordering assumptions are made).

This approach is fundamentally different from an actor system like Akka, where messages are directly addressed to actors and it is thus assumed that most messages arriving at an actor will in fact be handled by this actor. In this case, messages are always dispatched to the actor and matching happens on the receiving actor's thread, immediately followed by the execution of the associated handler code.

For Kompics Scala the requirement for handler matching was the ability to do full pattern matching on incoming messages. The simplest option to achieve this would have been to use Akka's approach and simply always schedule the component, and then match and immediately execute handler code, or, in other words, do all the work on the receiving thread. However, Kompics' broadcasting event dissemination model makes this exceedingly expensive in many applications, such as those with heavy use of a networking component that is shared between a large number of components. The alternative, then, was to match on the sending thread, as in Kompics Java, by linearly trying through all the subscribed `uponEvent` blocks of all ports reachable via channels from the origin port (the one the event was triggered on). However, if the right-hand side of any matched case-statement was executed immediately, as is the case in Akka, this approach would violate internal state isolation of the *receiving* component due to being executed on the *sending* thread. Thus, a level of indirection was needed between ascertaining whether or not the handler matches and actually executing the right-hand side of the case-statement. Scala's standard mechanism for this requirement would be `PartialFunction.isDefinedAt`. An implementation based on this mechanism would be very similar to the Java implementation, where the *sending* thread tests linearly through subscribed handlers until one (if any) is defined for the given event, then aborts and schedules the component. When the component is executed, the events are fully matched and the handler code gets immediately executed. A concern with this implementation would be that the imminent double matching could lead to large runtime overhead. This was already the case in Java, but in Scala it further involved potentially arbitrarily complex patterns. An alternative solution that avoids this issue, is to do the matching only once and remember the bound variables in a closure `() => Unit` capturing the handling code of the matched case-statement. This closure is then stored in a queue and later safely executed on the *receiving* component's thread, when scheduled. This approach also conveniently avoids the double matching being done in Java, which employs partial matching on the sending thread and full matching on the receiving one.

On the other hand, this technique pushes more computational work to the *sending* thread, and this can affect performance as we show in Section 5. There is clearly a performance tradeoff between the two described approaches depending on prevalent complexity of the matched patterns,

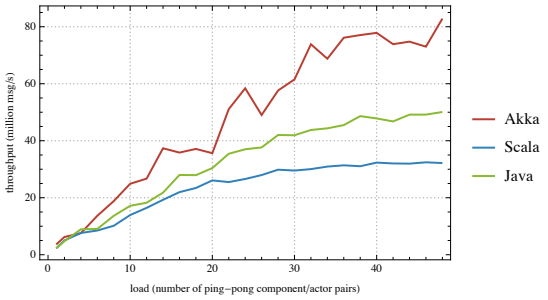

Figure 2. Matcher throughput comparison between Akka and Kompics Java, and Kompics Scala.

primarily driven by the cost of creating the closure compared to the work of extracting variables from the pattern. Exploratory benchmarks suggest that our approach typically starts outperforming the `isDefinedAt`-mechanism when a pattern matches an event with three or more fields, or a nesting depth of two or more patterns (e.g., extracting the value of a `Some` within an event case class). As Kompics Scala was written specifically with pattern matching in mind we assume that such complex patterns occur more often than not in practice and thus prefer the described solution. However, if this assumption should not hold in real code, an alternative `ComponentCore`/`PortCore` implementation with the `isDefinedAt`-mechanism would give developers the power to select their preferred approach.

Furthermore, we should also notice that the use of guard statements as part of the case offers a subtle pitfall, if mutable state of the *receiving* component is used, as this can lead to concurrency bugs due to execution of the guard statements on the *sending* thread.

5 Evaluation

We have performed an experimental evaluation of the performance impact introduced by Kompics Scala pattern-based matching approach, compared to Kompics Java and Akka. We have also used Kompics Scala with students and describe our experiences in this section.

5.1 Matching Performance

It was clear to us from the beginning that the approach of linearly pattern matching and then creating a closure, storing it, retrieving it on another thread and executing it, would cause some performance degradation compared to the simpler subtype-check-based Kompics Java model, or the even simpler schedule-match-and-execute-immediately model of Akka. In order to quantify this cost we decided to replicate an old Akka experiment by Nordwall [2012], which is effectively a "Ping-Pong" implementation[2] with

[2]Experiment source can be found at https://github.com/Bathtor/KompicsPerformanceTests

concurrency selectable by varying the number of ping-pong component/actor pairs.

Experimental Setup: The implementation for the experiments was done only in Scala, using the Java core for Kompics Java. The experiments were performed on a single Amazon AWS c4.8xlarge dedicated instance with 36-core Intel Xeon E5-2666 2.9Ghz CPUs and 60GB of memory. The Java runtime used was 64-Bit Oracle 1.8.0 on Ubuntu with similar configuration parameters to those used by Nordwall [2012]. Both Akka and Kompics were configured to use Java's `ForkJoinPool` as a scheduling backend with a parallelism of 8 and an Akka dispatcher throughput and Kompics η_{max} of 50. Each experiment was run with one billion messages and the time from start to end was measured in order to calculate the average throughput of the run.

Observations: As it can be seen in figure 2, Akka's simpler model, where almost all work happens on the receiving actor's thread, in general scales better than the Kompics model, which duplicates work and blocks the sending thread for some time to determine whether or not to schedule at all. On the other hand the tradeoff between Kompics Scala and Kompics Java turns out to be relatively fixed, with little to no cost in scalability, and a acceptable constant overhead likely caused by the additional object creation and management of the closure after matching.
It should, of course, be noted that this scenario in some sense is the best case both for Akka and Kompics Scala, as there are only two handlers, and all messages received are actually handled. A more practical scenario involving shared-network messages would likely provide a clearer view of the relative performance of these systems, but would not allow a clear distinction of the causes for the performance and was thus unsuitable for our current inquiry.

5.2 Educational Use

We have used Kompics Scala alongside Kompics Java in an on-premise course on distributed algorithms, which includes a project with the goal of a distributed linearisable key-value store based on the Paxos algorithm by Lamport [1998]. The students were given a choice of the implementation they wished to use, and also a project template in Kompics Java, which they were allowed, but not required, to base their code on. We shall consider the choice of Kompics Scala as an indicator of success for the implementation, if it was carried through the project (and not abandoned in favour of Java early on). Out of the 44 students who successfully completed the project ten chose to do the work in Kompics Scala, despite the extra effort of (partially) rewriting the project template. Out of those, two learned Scala just for this purpose during the course. Informal interviews after the course showed that the Scala users often had a much easier time translating the algorithms into practical code. The latter sentiment was echoed by many of the active students of the equivalent edX [edX 2012] course, which had around 5000 participants and was held completely in Kompics Scala, using Apache Zeppelin [Zep 2015] as an interactive coding and verification environment.

6 Conclusions

In this paper we have presented a Scala implementation of the Kompics component model that is compatible with a pre-existing Java implementation, while enabling full use of Scala's pattern matching features. We have described the DSL for this implementation and shown that it is much closer to a textbook representation of an algorithm, and additionally less verbose. We have also presented a novel way to schedule message handlers for pattern matching that allows a separation of matching thread vs. handler execution thread as is necessitated by Kompics' broadcasting. We have continued to evaluate the performance cost of this approach, and have discussed experiences in the context of distributed algorithms education with Kompics Scala. The new implementation has shown itself to be both useful and welcome in practice, with a manageable performance tradeoff.

References

2009. Akka. (2009). http://akka.io.

2009. Kompics. (2009). http://kompics.sics.se.

2012. edX. (2012). http://www.edx.org.

2013. CaracalDB. (2013). https://github.com/CaracalDB/CaracalDB.

2014. GVoD. (2014). https://github.com/Decentrify/GVoD.

2015. Apache Zeppelin. (2015). http://zeppelin.apache.org.

Cosmin Arad, Jim Dowling, and Seif Haridi. 2012. Message-passing Concurrency for Scalable, Stateful, Reconfigurable Middleware. In *Proceedings of the 13th International Middleware Conference (Middleware '12)*. Springer-Verlag New York, Inc., New York, NY, USA, 208–228. http://dl.acm.org/citation.cfm?id=2442626.2442640

Cosmin Ionel Arad. 2013. *Programming Model and Protocols for Reconfigurable Distributed Systems.* Ph.D. Dissertation. KTH - Royal Institute of Technology, Stockholm. https://www.kth.se/social/upload/51b05a6af276541b8120ce4d/CosminArad-PhD-Defence.pdf

Joe Armstrong. 2003. Making reliable distributed systems in the presence of software errors. December (2003), 295. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.408

Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. 2011. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media.

Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (may 1998), 133–169. https://doi.org/10.1145/279227.279229

Patrik Nordwall. 2012. 50 Million Messages Per Second - on a Single Machine. (2012). http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single-machine