Static Type Checking for the Kompics Component Model

Kola – The Kompics Language

Lars Kroll
Royal Institute of Technology (KTH)
lkroll@kth.se

Jim Dowling
Royal Institute of Technology (KTH)
jdowling@kth.se

Seif Haridi
Royal Institute of Technology (KTH)
haridi@kth.se

ABSTRACT

Distributed systems are becoming an increasingly important part of systems and applications software and it is widely accepted that writing correct distributed systems is challenging. Message-passing concurrency models are the dominant programming paradigm and, even in statically typed languages, programming frameworks typically only have limited type checking support for messages, channels, and ports or mailboxes. In this paper, we present Kola, a language-level implementation of Kompics, a component model with message-passing concurrency. Kola comes with its own compiler and some special language constructs which extend Java's type system as necessary to enforce static type checking on messages, channels, and ports. We show that Kola improves the readability of Kompics code and removes opportunities to introduce bugs, at the cost of little compile time overhead and no runtime overhead.

CCS Concepts

• **Software and its engineering** → **Distributed programming languages; Source code generation;** *Concurrent programming languages; Parsers;* • **Theory of computation** → *Distributed computing models;*

Keywords

component model, programming language, distributed languages, compilers, code generation, compile-time checks

1. INTRODUCTION

It is widely accepted that writing correct distributed systems is challenging, and thus any language or framework level support that is available to programmers can ease that burden. There are a number of programming languages available that are targeted specifically at concurrent and distributed programming, a prime example of which is Erlang[3]. Erlang's actor model with message-passing concurrency enforces programming patterns that make distributed execution of the Erlang code trivial. However, there is one important area where Erlang falls short, and that is at early bug detection through static type checking. Debugging distributed systems is generally a very difficult process, and any bug that can be caught before the code is ever run or deployed saves developers much time later. This fact has been recognised by the developers of the Akka[11] framework, which provides Erlang's actor message-passing concurrency model on top of Scala or Java, adding static types and the object-oriented paradigm[1]. However, traditionally Akka does not check statically whether the target of a message actually handles that type of message. The set of handled messages are rather defined as pattern matching rules, which are internal to an actor. While Akka avoids Erlang's indefinitely growing mailbox bug for unhandled messages, such a message is still almost always a bug and it would be beneficial to detect and prevent that bug at compile time already. There have been attempts (cf. in section 6 Typed Actors and Akka Typed) from the Akka developers to add this kind of static checking to Akka actors, but none have really seen widespread use so far.

There is an alternative to the actor model that has this event type checking built in, called the component model which has its roots in flow-based programming[8]. In this model messages travel along channels connected to ports, which specify exactly what type of messages are allowed to pass through them and in which direction. This allows for the specification of contracts in the form of message types on ports, which can be enforced at compile time. Kompics is a message[2]-passing component model [1, 2] and is implemented in Java as a framework. The Java compiler, however, does not allow for all the checks we would like to do at compile-time as some of the information is expressed in a domain specific language (DSL) that cannot be expressed completely in Java's type system.

In this paper we present Kola, a language implementation of the Kompics component model. It extends the Java language with special language constructs for events, ports, components, and handlers, which are explained in section 3. The Kola compiler, described in section 4, transforms these special constructs back into normal Java source code while simultaneously performing the type checks on events, channels, and ports, which the Java compiler cannot do. The Java source code generated by Kola is then compiled by the Java compiler to run on the Java Virtual Machine (JVM).

[1]In case of Java it also removes the functional paradigm as found in Erlang.

[2]called *event* in Kompics

PMLDC '16, July 17 2016, Rome, Italy

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4775-4/16/07. . . $15.00

DOI: http://dx.doi.org/10.1145/2957319.2957371

```java
public class PortP extends PortType {{
        indication(EventA.class);
        request(EventB.class);
}}
public class ComponentC extends ComponentDefinition {
    Positive<PortP> pp = requires(PortP.class);
    Handler bHandler = new Handler<EventB>(){
        public void handle(EventB event) {
            trigger(new EventA(), pp);  // error #3
        }
    };
    {
        subscribe(bHandler, pp);  // error #4
    }
}
public class ComponentD extends ComponentDefinition {
    Handler<Start> startHandler = new Handler<Start>(){
        public void handle(Start event) {
            // Do something
        }
    }; // warning #1
}
public class ParentComponent extends ComponentDefinition {{
  Component cc = create(ComponentC.class, Init.NONE);
  Component cd = create(ComponentD.class, Init.NONE);
  connect( cc.getPositive(PortP.class),  cd.getNegative(PortP.class) ); // errors #1 & #2
  // warning #2 as a consequence of the errors above
}}
```

Listing 1: Pedagogical example with all Kompics mistakes the Java compiler cannot catch.

The resulting JVM bytecode is fully interoperable with any other Java code, and in particular with Kompics code from the Java library implementation. In section 5, we compare Kola and Kompics' Java implementation to show that Kola improves the readability of Kompics code and reduces the opportunities to introduce bugs, at the cost of some compile time overhead and almost no run time overhead.
Section 6 introduces related work, and we conclude by identifying opportunities for future work in section 7.

2. BACKGROUND

2.1 The Kompics Component Model

Since it is necessary to understand the Kompics component model to understand the contributions of Kola, we begin with a brief overview of Kompics, and a more detailed overview can be found in [2].

Semantics.

Kompics is a programming model for distributed systems that implements protocols as event-driven *components* connected by *channels*. Kompics provides a form of type system for events, where every component declares its required and provided *ports* – they can be thought of as "services" –, which in turn 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` can 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` (or their subtypes) 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.

Scheduling.

A component is guaranteed to be only scheduled on one 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, and keeps handling events until either there are no more events queued at its ports or a configurable maximum number of events to be handled is reached. After the component has finished handling events, it will be placed at the end of the FIFO queue of components waiting to be scheduled. Tuning the configurable maximum number of events to be handled 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.

Event-handling.

In contrast to Actor systems like Akka [11] or Erlang [3], 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 *handler*s on their declared ports. Note that ignored messages are silently dropped, which is necessitated by the channel broadcasting model, that is to say, as opposed to Erlang and Akka, in Kompics it is often completely correct to simply ignore a large number of events.
Matching of events to handlers is based on the events' type-

hierarchy, although there are some Kompics extensions that provide pattern matching as well.

Note that in addition to the default Java library implementation of Kompics, there are alternatives written in Scala and Python as well, which are not considered in this paper.

2.2 Kompics Java Framework Limitations

There are a number of issues that commonly arise in programming with Kompics in Java, which could be checked at compile time by a compiler with domain specific knowledge. Listing 1 shows a pedagogical example, where we put all these issues into as little code as possible. We classify the issues into *errors*, and *warnings*.

Errors.

are issues that will result in immediate system termination upon detection at run time. We consider here the following errors that can be avoided at compile time, i.e. they should prevent successful termination of the compilation process.

1. *Connecting a component on a port it does not require or provide.*
   If a component does not declare a port, it can, of course, not be connected on that port. But since the lookup of declared ports happens at run time, this is only caught at that late stage. An example of this can be seen in line 26 of listing 1, where `cd` gets connected on a required port `PortP` which its definition `ComponentD` never declares (cf. lines 16-22).

2. *Connecting the ports of two components the wrong way around.*
   Similar to the above case, just that this time the component does declare the port to be connected but in the other direction. For example, in line 26 the call `cc.getPositive(PortP.class)` requests a provided port of type `PortP`, but `ComponentC`, which is the definition for `cc`, only declares a required `PortP` (line 6).

3. *Triggering events on ports which do not carry them (in that direction).*
   Since indication and request declarations in the port types define which direction events are allowed to go, it can happen in asymmetrical port types – which are the vast majority of port types – that it is attempted to trigger an event on a port, which only goes the other way. An example of this can be seen in line 9, where an instance of `EventA` is triggered on `pp` which is a required port of type `PortP`. On a required port only requests can be triggered, but `EventA` is an indication on `PortP` (cf. line 2).

4. *Subscribing an event handler to a port for an event that is not carried on that port (in that direction).*
   The same issue as above can arise with handler subscriptions as well. Handlers need to handle (subtypes of) events that are triggered on the external side of the ports they are subscribed to, thus exactly the opposite set of what can be triggered from inside a component. In line 13 a handler for `EventB` (line 7-11) is subscribed to `pp`, which is, as before, a required port of type `PortP`. Thus only requests can be handled on such a port, but `EventB` is an indication in `PortP` (cf. line 3).

Warnings.

are issues, which should not terminate the run time system, because they could be desired behaviour under certain circumstances, but the following specific ones quite often lead to difficult to find bugs, making it valuable for a compiler to point them out.

1. *Creating an event handler without subscribing it to a port.*
   This is by far the most common bug in Kompics code that regularly happens to both novices and seasoned programmers. Most people separate handler creation, which is grouped with class fields, and handler subscription, which goes into a constructor or instance initialiser, making it very easy to overlook this issue even on repeated code reviews. At run time, this issue manifests itself simply as lost events, which are frustratingly difficult to pin-point (Was the event even sent? Are all channels along the way properly connected? Did it get filtered out somewhere?). For example, in `ComponentD` (lines 16-22) the `startHandler` is never subscribed, but it might look at run time as if the component is never started.
   In 95% of all cases not subscribing a handler immediately is a bug. However, in the remaining 5% it is completely reasonable, sometimes even necessary, behaviour. For example, in a Kompics component that implements a finite state machine (FSM) it would be very typical to subscribe and unsubscribe handlers dynamically, upon state changes. These handlers might even be contained in some kind of collection, to group them according to the state(s) they belong to, making it very difficult for any compiler to keep track of whether they are ever subscribed.

2. *Creating components without connecting all their required ports.*
   This issue comes up significantly less often than forgetting to subscribe handlers, mostly because component creation and port connections are usually done in the same block of code. But it can happen, especially in large initialisation components with many children, that a port is not connected properly, which at run time manifests itself again as silently lost events. This happens implicitly in line 26 again, as we are not connecting the ports correctly, but in this case it would not manifest itself, as the run time would already abort due to the errors in that line (see above).
   Yet again, in a few cases this might in fact be desired behaviour, introduced by initialisation order dependencies, or dynamic service provider exchange with the required components.

All the issues described above were the driving force behind the inception of Kola and are, at least to some degree, addressed by the language's design and handled by its compiler.

3. KOLA GRAMMAR

This section describes the new keywords and syntactical structures with which *Kola*, the *Kompics language*, extends Java, and how it is mapped to Java code. Where it does not conflict with the syntax mentioned here, Kola code is

```
event SingletonEvent
```

is compiled to

```
public class SingletonEvent implements KompicsEvent {
    public final static SingletonEvent event = new
         SingletonEvent();
    private SingletonEvent() {}
}
```

and

```
event TestEvent(String some)
```

is compiled to

```
public class TestEvent implements KompicsEvent {
    public final String some;
    public TestEvent(final String some) {
        this.some = some;
    }
}
```

Figure 1: Examples of event declarations.

simply normal Java code. Syntax definitions are collected in appendix B[3].

3.1 Keywords and Tokens

Kola introduces the following new keywords and tokens:

> **handler**, **handle**, **port**, **component**, **componentdef**, **!subscribe**, **!unsubscribe**, **!connect**, **!disconnect**, **init**, **!trigger**, **requires**, **provides**, **indication**, **request**, **event**, **=>**

For backwards-compatibility reasons all new keywords that can be used as part of a statement are prefixed with an exclamation mark (`!`), in order to avoid confusing the parser when parsing the equivalent method names from the Kompics framework. The intention behind choosing `!` as a prefix was to allude to Erlang's and Akka's message sending "tell" operator.
Since these keywords hide some identifier tokens from Java Kompics code, a few special rules have been introduced, to allow reinterpretation of the keywords as identifiers in certain contexts, such as method invocation and field references.

3.2 Events

Kola provides a new syntax to declare *events*, shown in figures 1 and 7, designed to reduce verbosity and repetitive code required for such pure immutable data classes.
The syntax for event declarations is inspired by Scala's *case classes*[9], in that they allow specification of public fields in the header of the class description. If the event description has no header fields, it is assumed that a *case object* (Scala terminology) is requested and a *singleton event* is created, the only instance of which can be accessed using the **.event** accessor, similar to Java's **.class** accessor.
Note that the generated Java class (see figure 1) always implements Kompics' event marker interface `KompicsEvent` and by default all header fields are mapped to public final fields, as this is the recommended practice in Kompics.

[3]The full grammar in Sablecc format can be found at: https://github.com/kompics/kola/blob/master/src/main/sablecc/kola-0.1.7.sablecc

```
port TestPort {
  indication { TestEvent1, TestEvent3 }
  request { TestEvent2 }
}
```

is compiled to

```
public class TestPort extends PortType {
    {
        indication(TestEvent1.class);
        indication(TestEvent3.class);
        request(TestEvent2.class);
    }
}
```

Figure 2: Examples of port declarations.

```
componentdef TestC {
  init(String some) {
    //do something with some
  }
}
```

is compiled to

```
public class TestC extends ComponentDefinition {
    TestC(String some) {
        //do something with some
    }
    public TestC(final AutowireInit1 autowireInit) {
        this(autowireInit.some);
    }
    public static class AutowireInit1 extends
         Init<TestC> {
       public final String some;
       public AutowireInit1(String  some) {
            this.some = some;
        }
    }
}
```

Figure 3: Examples of component definitions.

3.3 Port Types

A port type is effectively a singleton class that maps lists of event types to the directional groups: indications and requests. The Kola syntax for port types (figures 2 and 8) is rather similar to the Java syntax (cf. figure 2), merely avoiding one initialisation block, the **.class** accessor for each event type, and allowing events to be grouped in a more obvious manner. Most importantly, the introduction of port type declarations into the grammar makes it easier for the compiler to recognise them, and handle them appropriately during type checks later when these types are used in components.

3.4 Component Definitions

Component definitions are the templates for components, in much the same way that classes are templates for objects. They define the ports, handlers, child components, and internal state that a component instance of a specific type will have.
The component definition syntax (figures 3 and 9) allows

```
componentdef TestC {
  provides TestPort test;
  requires TestPort2 test2;
}
```

is compiled to

```
public class TestC extends ComponentDefinition {
  protected final Negative<TestPort> test =
       provides(TestPort.class);
  protected final Positive<TestPort2> test2 =
       requires(TestPort2.class);
}
```

Figure 4: Examples of port fields.

```
1 componentdef TestParent {
2   component TestC("someString") child;
3   component TestC dynamicChild;
4 }
```

is compiled to

```
1 public class TestParent extends ComponentDefinition {
2   protected final Component child =
         create(TestC.class, new
         AutowireInit1("someString"));
3   protected Component dynamicChild;
4 }
```

Figure 5: Examples of child component declarations.

everything in a body that a normal class definition would allow, plus additional Kompics declarations of init-blocks, ports, children, handlers and handler-related statements.

Init Blocks.

Component instances are created by the Kompics runtime, not directly by the programmer, and thus configuration parameters to components must be passed in an `Init` object.[4] However, creation of such init objects adds a certain amount of boilerplate in Java, in terms of mapping constructor arguments to fields. Kola introduces the **init**-block (figures 3 and 12) to avoid this boilerplate. Writing an **init**-block in Kola compiles to a constructor and an `AutowireInit` class of the correct arity in Java (cf. figure 3).

Port Fields.

Every component provides and requires a number of named ports of a specific type. In Java these are simply variables that are initialised once the component is loaded, but in Kola they are treated as a special type of field by the compiler, identified by the **provides** and **requires** keywords (figures 4 and 10)

Children.

Components in Kompics form a supervision hierarchy, similar to Erlang actors. Child components can be created dynamically at runtime or statically at component creation time. Kola only deals with static creation at this time (figures 5 and 11), while dynamic creation works as before in the Java version, i.e. using the `create` method of the parent component definition.

[4]In old Kompics version this was done via an Init-event, but this method was unnecessarily complicated and was optimised away.

Both

```
componentdef TestC {
  provides TestPort test;
  handle testHandler => test : TestEvent e {
    // do something with e
  }
}
```

and

```
componentdef TestC {
  provides TestPort test;
  handler testHandler : TestEvent e {
    // do something with e
  }
  !subscribe testHandler => test;
}
```

are compiled to

```
public class TestC extends ComponentDefinition {
  protected final Negative<TestPort> test =
       provides(TestPort.class);
  protected final Handler<TestEvent> testHandler
     = new Handler<TestEvent>() {
        public void handle(TestEvent e) {
            testHandlerMethod(e);
        }
    };
    {
        subscribe(testHandler, test);
    }
    private final void testHandlerMethod(TestEvent
         e) {
        // do something with e
    }
}
```

Figure 6: Examples of handlers declarations.

If the component initialisation part is omitted (figure 5 line 3), only a child component field is created, but the component is not initialised or started automatically, leaving it prepared for dynamic instantiation.
If the argument list is empty, the component is initialised with a no-argument constructor, i.e. equivalent to using `Init.NONE`.
If arguments are provided, the Kola compiler will attempt to find a matching `AutowireInit` class (see paragraph on *Init Blocks* in section 3.4) for this component and use that for initialisation.

Handlers.

are the parts of a component that deal with incoming events. Handlers are subscribed to ports. Subscription can be done statically or dynamically, as was the case with child component creation. However, in this case Kola supports both (figures 6 and 13). A **handler** declaration simply declares a special type of field that holds a handler object (an implementation of the `Handler` interface from the Kompics framework). A handler does not do anything unless it is subscribed to a port, which can be done with the **!subscribe** statement and undone with the **!unsubscribe** statement. Kola allows these statements directly in the component definition body, and will compile them into the instance initialiser in Java. Since forgetting to subscribe a handler to a port is one of the most common mistakes in Kompics in practice (cf. section 2.2 warning 1), Kola also defines a **handle** declaration, which does handler creation and subscription in one step. This should be the default approach of creating handlers in Kola. The handler can still be unsubscribed later, if so desired.

Note that the extra method `testHandlerMethod` in figure 6 lines 12-14 is not technically necessary. The reason that Kola compiles to that form instead of simply doing the work in the handler's **handle** method, is that it feels very unintuitive to use `TestC.this` when trying to access a shadowed parent component field in this format. This is avoided by invoking an unshadowed method of the parent object.

Channels.

are created using the **!connect** statement, and destroyed using the **!disconnect** statement. As with subscriptions this can be done both statically and dynamically (figure 14). Both statements take the *provided* port to the left of the arrow (`=>`) and the *required* port to the right.

Generally, this is compiled in the obvious manner by simply replacing the keyword and `=>` with equivalent binary method from the Kompics framework. However, if the type of either of the expressions is not a port type, instead of using the expression directly a `.getPositive(PortType.class)` or `.getNegative(PortType.class)` is suffixed to the expression.

3.5 Trigger

In Kompics events are triggered on ports inside a component and then forwarded along all the connected channels. Kola also adds a **!trigger** statement (figure 15), mostly for consistency with the other new statements, but also to point out to the compiler that it should check for error 3 (cf. section 2.2).

The type of the left expression can be any event type, the type to the right has to be a port instance.

3.6 Other

As a small extension to Java, Kola allows multiple top-level type declarations in a single file, similar to Scala[9]. This makes it possible to easily keep a component or a port type with all its associated events together for easy reference, without having to resort to static inner classes.

Kola also keeps all classes in `se.sics.kompics` permanently in scope in addition to Java's `java.lang`. The necessary imports for the Java source files are automatically generated by the compiler.

Example.

A longer Kola code example can be found in appendix C.

4. COMPILER

The Kola compiler, `kolac`, consists of four major parts: A lexer, a parser, an abstract syntax tree (AST) analyser/-transformer, and a Java source generator and writer. Additionally, there is an Apache Maven[5] plugin that allows for Kola projects to be compiled in a way that is familiar to Java programmers.

[5]http://maven.apache.org/

4.1 Lexer & Parser

Both the Lexer and the Parser are automatically generated by the SableCC[4] from a grammar file[6]. The file is based on an older Java 1.7 grammar provided on the SableCC website[7]. It was extended with the Kola specific rules described in section 3 and transformation rules from concrete syntax tree (CST) to abstract syntax tree (AST) were added, as they are supported in newer SableCC versions.
Lexing and parsing is done on multiple source files in the `sourcePath` in parallel, since this stage is trivially parallelisable. If no errors occur during lexing and parsing, the AST analysis stage is invoked for all source files in sequence to avoid concurrency issues with type resolution.

4.2 AST Analyser/Transformer

A one-pass depth first approach is used to analyse the AST, and transform it to a Java syntax tree (JST). The format used for the JST is a modified version of Sun's JCodeModel[8]. Where names cannot be immediately resolved, markers are inserted into the JST, to be resolved later after the AST$\rightarrow$JST pass has finished, and all source files have been analysed. Furthermore, in this stage the new Kola grammar structures are resolved into Java structures, and either implicitly or explicitly checked for compliance with Kompics' semantical restrictions (cf. section 2.2).

4.3 Java Source Generation

This stage simply uses JCodeModel's source generation facilities and a file writer, to generate the right directory structure and Java source files for the JST in the `outputDirectory`. During this stage it is also attempted to resolve previously unresolved names in a lazy manner, and if necessary errors are thrown where this is still not possible.

4.4 Apache Maven Plugin

The Maven plugin wraps the actual `kolac` executable, collects the correct `CLASSPATH` elements, replaces the normal Maven `sourceDirectory`(s) with the `kolac outputDirectory` and compiles all the Kola and Java files in the `kolac inputDirectory`. It is run in the `generate-sources` phase of the Maven build lifecycle.

5. EVALUATION

In order to be a valuable addition to the Kompics ecosystem, Kola had to fulfil the following three criteria, which we will evaluate in this section:

1. Reduce opportunities for mistakes that are either unnecessarily only caught at run time or lead to difficult to find bugs.

2. Improve the readability of Kompics code.

3. Introduce as little overhead as possible, especially at run time.

5.1 Mistakes that are Preventable at Compile Time

[6]http://github.com/kompics/kola/blob/master/src/main/sablecc/kola-0.1.7.sablecc
[7]http://www.sablecc.org/java1.7/
[8]http://github.com/Bathtor/JCodeModel

```java
public class HelloC extends ComponentDefinition {{
        Handler startHandler = new Handler<Start>() {
            @Override
            public void handle(Start event) {
                System.out.println("Hello World from
                    Kola!");
                Kompics.asyncShutdown();
            }
        };
        subscribe(startHandler, control);
}}
```

Listing 2: Kompics Hello World (omitting the `Main` class used to start the runtime).

Project	LoC	Java	Kola
Hello World	33	2	0
Ping Pong	124	12	0
CaracalDB	37711	894	5
GVoD	94000	1358	0

Table 1: Number of opportunities for mistakes that are preventable at compile time in Java and Kola.

Opportunities for mistakes that are preventable at compile time are described in section 2.2 and broadly fall into the categories of *errors* and *warnings*. Since there is no formal measure of this property, we shall simply resort to counting such opportunities in a number of example programs. As described above, these opportunities arise in the following circumstances:

- **`connect`** method calls.
- **`trigger`** method calls.
- **`subscribe`** method calls.
- `create` method calls.
- `Handler` instance creation.

Thus our metric is the total number of occurrences of these circumstances in the example program.
We will use example programs ranging from a very simple *Hello World* (listing 2), over a more involved *Ping Pong*[9], to multi-thousand line projects, namely *CaracalDB*[10] and *GVoD*[11].

For the *Hello World* (listing 2) example, it is easy to see, that there are exactly two such opportunities: The creation of the `startHandler`, and the **`subscribe`** invocation. The larger projects were, of course, not counted manually, but rather using common bash tools.

It can be seen from table 1, that the the number of such opportunities in large projects grows sublinearly, possibly loosely logarithmically, with the size of the code base. That is to say it levels off for larger projects, where pure Java code starts to become dominant over the distributed aspects. Thus it is clear that the impact of having these opportunities removed is very high, especially in the early stages of a new Kompics project.

It should be noted, as can be seen in the *CaracalDB* entry in table 1, that the Kola compiler currently does not in fact remove all such issues that could theoretically be detected at compile time. If handlers or components are created and placed in some kind of container data structure, for example a collection, before subscriptions or connections respectively have been done for the variable, the Kola compiler loses track of these objects. However, this is mostly an issue for the two *warnings*. *Errors* could theoretically run into the same problem, but only in Kola code that a) purposefully ignores the Kola constructs in favour of their Java equivalents, and b) is additionally very badly structured, for example by moving required ports into collections, or purposefully casting their types, such that the port type information is lost. This possibility cannot be avoided while maintaining backwards compatibility with the Kompics Java implementation.

5.2 Readability

Code readability is not trivial to measure either, but what we are really looking for here are *software productivity metrics*. The most common productivity metric by far is lines of code (LoC). However, LoC alone is not suitable as a comparison between languages, as it does not take the number of keywords, i.e. the *vocabulary*, of the language into account, which is a major factor in software productivity.
This is captured, among other things, by the Halstead metrics[5], which we will use here to compare Kola and Kompics Java. From these we used the *length* (N), which is the total number of operands and operators in the program, the *vocabulary* (n), the number of distinct operands and operators, the *volume* (V), which captures the relationship between length and vocabulary ($N \times \log_2 n$), as well as the *difficulty* (D) to write or understand the code, and the development *effort* (E), which translates into a coding *time* (T) approximation of $T = \frac{E}{18}$ seconds. It should be noted that the Halstead metrics are originally designed for procedural code, and do not map trivially to object oriented languages like Java or Kola. However, when adding support for them to the Kola compiler we opted for a rather naïve implementation, as the goal was not to actually predict development time accurately, but only to compare Kola and Java Kompics, and since both use the same implementation, the comparison should be rather fair.

Another commonly used software productivity metric is Cyclomatic Complexity (CC)[7], which measures the complexity of programs based on their control flow structure. However, CC is more useful to measure good coding practices within a language, than improvements across languages, and will thus not be used here.

Since there are no large projects available, yet, for Kola, we had to limit our evaluation with software productivity metrics to smaller programs that could quickly be written in both Kola and Kompics Java to be as similar as possible. All the examples used for table 2 can be found in the *Kola Examples* repository[12] of which listing 2 is an excerpt. All the metrics, both for Kola and Java code, collected for this section were calculated by the Kola compiler.

Table 2 shows lines of code and the Halstead metrics mentioned above for the *Hello World* (HW) and *Ping Pong* (PP)

[9]http://github.com/kompics/kola-examples/tree/master/src/main/java/se/sics/kola/examples/pingpong/java
[10]https://github.com/CaracalDB/CaracalDB
[11]https://github.com/Decentrify/GVoD

[12]https://github.com/kompics/kola-examples

	LoC		N		n		V		D		E $\times 10^3$	
Project	Java	Kola	Java	Kola	Java	Kola	Java	Kola	Java	Kola	Java	Kola
Hello World	33	14	151	60	48	38	843	315	22	8	18.6	2.5
Ping Pong	124	62	635	259	105	89	4263	1677	69	38	295	63

Table 2: Software productivity metrics for example programs in Kola and Kompics Java. Lines of Code (LoC), and Halstead metrics: Length (N), Vocabulary (n), Volume (V), Difficulty (D), Effort (E).

	LoC		raw		adj.	
Project	Java	Kola	Java	Kola	Java	Kola
Hello World	33	14	1.579	2.181	0.506	1.108
Ping Pong	124	62	1.681	2.381	0.608	1.308

Table 3: Average compile time in seconds as reported by Maven and adjusted for Maven overhead of 1.0728s.

examples. It can be seen that in both Java and Kola PP is about four times longer than HW, which is reflected both in LoC and in N (the length). The vocabulary, however, only doubles between the two programs. What is also immediately obvious, is that Java code is about twice as long as Kola code, while having only about 20% more vocabulary. Both the overhead in length and vocabulary, translate directly to large differences in difficulty (D) and development effort (E), where both metrics show almost an order of magnitude overhead of Java over Kola.

5.3 Overhead

Two types of overhead are of interest for us: Compile time overhead, introduced by our additional Kola compiler, and run time overhead, which could occur if the Kola compiler produces less efficient code, compared to hand written Kompics code.
The experiments in this section were run on a MacBook Pro with a 3GHz dual-core Intel Core i7 (L2 Cache 256KB per core, L3 Cache 4MB) and 16GB of DDR3 memory, as well as a solid-state drive (SSD). The JVM used was Oracle Java HotSpot(TM) 64-Bit Server VM version `1.8.0_31` for OS X.

Compile Time Overhead.

It is clear that there has to be *some* compile time overhead, as extra – partially redundant – work is being done. The question we have to answer is how much, and how does it scale. We have measured the compile time for small, comparable Kompics Java and Kola projects, and to make these results more realistic we used the Maven plugin, instead of measuring raw compiler time. The values for Kola include both the `kolac` and `javac` execution times, while Kompics Java only measured `javac`. We additionally measured Maven overhead, by compiling a completely empty project with the same setup.
Table 3 shows the results for projects from the *Kola Examples* repository (see above), averaged over five measurements each. As can be seen easily in the adjusted columns, the Kola compiler adds about the same compilation effort again as the Java compiler, which seems rather reasonable as it does similar work to the latter and is probably somewhat less optimised.

Run Time Overhead.

As opposed to compile time overhead, it is not clear whether the Kola compiler introduces any noticeable inefficiencies into the generated code. However, the only performance critical aspects that are different in Kola are handler definitions, and thus any introduced overhead should be reflected in a loss of raw event handling throughput. We used the *Ping Pong* example to collect event throughput statistics and compared the performance of the Kompics Java and the Kola implementations. The experiments were run using the Kompics scheduler with 2 threads and an event batch size of 50 events. The JVM used default settings, as memory management was not an issue, since no new objects were being created during the measurement phase. For five consecutive one minute runs each the Java version reached an average of $1,117,663$ events per second ($\sigma = 46244$), and the Kola version showed an average of $1,084,100$ events per second ($\sigma = 95250$). This leaves both results within one standard deviation (σ) of each other and thus does not conclusively show overhead of one over the other, but possibly indicates a slight slowdown of the Kola version. This might be caused by the introduction of the additional method call in handlers compiled from Kola to Java (cf. section 3.4 the paragraph on *Handlers*).

6. RELATED WORK

Distributed Oz[10] and Erlang[3] are the most well-known examples of programming languages with a clear distributed systems focus, yet both languages are dynamically typed.
Akka[11] on Scala and Java endeavours to bring static typing to the actor model used in Erlang, but usually does not check the types of messages that are sent to specific actors. There have been mutliple attempts to introduce type-safety in Akka at that level: 1) *Typed Actors* are an implementation of the *Active Objects* pattern on top of Akka, effectively hiding the message-passing model below intercepted method invocations. This approach, however, tends to be very unintuitive and has never reached widespread application among Akka users. 2) *TAkka*[6] supports statically typed messages, and actor behaviours. 3) *Akka Typed*[13], is an experimental addition to Akka that decouples *behaviour* specifications and the actors that exhibit such behaviour. This is very similar to the Kompics model, replacing channels and events with addressed messages.
The Go[14] language was created by R.Griesemer, R.Pike, and K.Thompson at Google and is a statically typed language that uses statically typed channels for concurrency.
The Scala DSL for Kompics[15] addresses some of the ver-

[13]Akka Typed was presented by R.Kuhn at CurryOn in July 2015
[14]http://golang.org
[15]http://kompics.sics.se/current/scala/

bosity and readability issues of Kompics Scala that are measured in section 5.2, but does not add any new type checks. Quite the opposite, in fact, it removes some of the handler types in favour of Akka-like pattern matching.

7. SUMMARY AND FUTURE WORK

In this paper we have introduced Kola, the Kompics language, a Java 1.7 extension that brings static type checking to the domain of events in component based systems. We have shown which kind of issues typically occur in Kompics Java that can be prevented at compile time already by Kola, and how frequently the opportunity for these kinds of bugs arises. We have also shown a large improvement in code readability for Kola, as measured by common software productivity metrics. And finally we have shown that the compile time overhead introduced by Kola is manageable, while the run time overhead is negligible to non-existent.
We are planning to extend the Kola grammar to support Java 1.8, as that version, among other things, allows shorter handler code in Kompics Java using anonymous functions, making the comparison between Kola and Java a fairer one.
We are also planning to incorporate feedback and experiences from students being taught distributed systems (ID2203) at KTH in Kola instead of Kompics Java into the future development of the language.
Formal definition of the type checking rules the Kola compiler uses, where they differ from the Kompics rules described in [2], is also left for future work.

APPENDIX

A. REFERENCES

[1] C. Arad, J. Dowling, and S. Haridi. Message-passing Concurrency for Scalable, Stateful, Reconfigurable Middleware. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 208–228, New York, NY, USA, 2012. Springer-Verlag New York, Inc.

[2] C. C. I. Arad. *Programming Model and Protocols for Reconfigurable Distributed Systems*. PhD thesis, KTH - Royal Institute of Technology, Stockholm, 2013.

[3] J. Armstrong. Making reliable distributed systems in the presence of software errors. (December):295, 2003.

[4] E. Gagnon and L. Hendren. SableCC – an object-oriented compiler framework. *Proceedings of TOOLS 1998*, 1998.

[5] M. H. Halstead. *Elements of software science*, volume 7. Elsevier New York, 1977.

[6] J. He, P. Wadler, and P. Trinder. Typecasting Actors: From Akka to TAkka. *Proceedings of the Fifth Annual Scala Workshop*, pages 23–33, 2014.

[7] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[8] J. P. Morrison. Data responsive modular, interleaved task programming system. *IBM Technical Disclosure Bulletin*, 13(8):2425–2426, 1971.

[9] M. Odersky. The Scala experiment: Can we provide better language support for component systems? *Proceedings of the 2006 POPL Conference*, 41(1):166–167, 2006.

[10] P. V. A. N. Roy, G. Smolka, M. Mehl, R. Scheidhauer, P. Van Roy, S. Haridi, and P. Brand. Mobile Objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, 1997.

[11] D. Wyatt. *Akka Concurrency*. Artima Incorporation, USA, 2013.

B. SYNTAX DEFINITIONS

⟨*EventDeclaration*⟩ ::= [⟨*Modifiers*⟩] '`event`' ⟨*Identifier*⟩ [⟨*TypeParameters*⟩] [⟨*HeaderFields*⟩] ['`extends`' ⟨*ClassType*⟩] [⟨*Interfaces*⟩] [⟨*ClassBody*⟩]

⟨*HeaderFields*⟩ ::= '`(`' [⟨*FormalParameterList*⟩] '`)`'

Figure 7: Syntax of event declarations.

⟨*PortDeclaration*⟩ ::= [⟨*Modifiers*⟩] '`port`' ⟨*Identifier*⟩ ⟨*PortBody*⟩

⟨*PortBody*⟩ ::= '`{`' ⟨*PortBodyDeclaration*⟩* '`}`'

⟨*PortBodyDeclaration*⟩ ::= '`indication`' '`{`' ⟨*EventType*⟩* '`}`'
| '`request`' '`{`' ⟨*EventType*⟩* '`}`'

Figure 8: Syntax of port declarations.

⟨*ComponentDeclaration*⟩ ::= [⟨*Modifiers*⟩] '`componentdef`' ⟨*Identifier*⟩ [⟨*TypeParameters*⟩] ⟨*ComponentBody*⟩

⟨*ComponentBody*⟩ ::= '`{`' ⟨*ComponentBodyDeclaration*⟩* '`}`'

⟨*ComponentBodyDeclaration*⟩ ::=
⟨*ClassMemberDeclaration*⟩
| ⟨*InstanceInitializer*⟩
| ⟨*StaticInitializer*⟩
| ⟨*ConstructorDeclaration*⟩
| ⟨*InitDeclaration*⟩
| ⟨*PortFieldDeclaration*⟩
| ⟨*ChildDeclaration*⟩
| ⟨*HandlingDeclaration*⟩

Figure 9: Syntax of component definitions.

⟨*PortFieldDeclaration*⟩ ::= '`requires`' ⟨*PortType*⟩ ⟨*Identifier*⟩ '`;`'
| '`provides`' ⟨*PortType*⟩ ⟨*Identifier*⟩ '`;`'

Figure 10: Syntax of port field declarations.

$\langle$*ChildDeclaration*$\rangle$ ::= ‘`component`’ $\langle$*ComponentType*$\rangle$ [$\langle$*ComponentInitialization*$\rangle$] $\langle$*Identifier*$\rangle$ ‘`;`’

$\langle$*ComponentInitialization*$\rangle$ ::= ‘`(`’ [$\langle$*ArgumentList*$\rangle$] ‘`)`’

Figure 11: Syntax of child component field declarations.

$\langle$*InitDeclaration*$\rangle$ ::= [$\langle$*Modifiers*$\rangle$] ‘`init`’ [$\langle$*HeaderFields*$\rangle$] $\langle$*ConstructorBody*$\rangle$

Figure 12: Syntax of init block declarations.

$\langle$*HandlingDeclaration*$\rangle$ ::= $\langle$*HandleDeclaration*$\rangle$
| $\langle$*HandlerDeclaration*$\rangle$
| $\langle$*ConnectStatement*$\rangle$
| $\langle$*SubscribeStatement*$\rangle$
| $\langle$*DisconnectStatement*$\rangle$
| $\langle$*UnsubscribeStatement*$\rangle$

$\langle$*HandleDeclaration*$\rangle$ ::= ‘`handle`’ $\langle$*Identifier*$\rangle$ ‘`=>`’ $\langle$*PortIdentifier*$\rangle$ ‘`:`’ $\langle$*EventType*$\rangle$ $\langle$*EventIdentifier*$\rangle$ $\langle$*Block*$\rangle$

$\langle$*HandlerDeclaration*$\rangle$ ::= ‘`handler`’ $\langle$*Identifier*$\rangle$ ‘`:`’ $\langle$*EventType*$\rangle$ $\langle$*EventIdentifier*$\rangle$ $\langle$*Block*$\rangle$

$\langle$*SubscribeStatement*$\rangle$ ::= ‘`!subscribe`’ $\langle$*HandlerRef*$\rangle$ ‘`=>`’ $\langle$*PortRef*$\rangle$ ‘`;`’

$\langle$*UnsubscribeStatement*$\rangle$ ::= ‘`!unsubscribe`’ $\langle$*HandlerRef*$\rangle$ ‘`=>`’ $\langle$*PortRef*$\rangle$ ‘`;`’

Figure 13: Syntax of handler related declarations.

$\langle$*ConnectStatement*$\rangle$ ::= ‘`!connect`’ $\langle$*Expression*$\rangle$ ‘`=>`’ $\langle$*Expression*$\rangle$ ‘`:`’ $\langle$*PortType*$\rangle$ ‘`;`’

$\langle$*DisconnectStatement*$\rangle$ ::= ‘`!disconnect`’ $\langle$*Expression*$\rangle$ ‘`=>`’ $\langle$*Expression*$\rangle$ ‘`:`’ $\langle$*PortType*$\rangle$ ‘`;`’

Figure 14: Syntax of connect statements.

$\langle$*TriggerStatement*$\rangle$ ::= ‘`!trigger`’ $\langle$*Expression*$\rangle$ ‘`=>`’ $\langle$*Expression*$\rangle$ ‘`;`’

Figure 15: Syntax of trigger statements.

C. KOLA EXAMPLE

The following example is very similar to the *Ping Pong* code used for section 5, with the performance measuring code removed.

```
event Ping
event Pong

port PingPongPort {
  indication { Pong }
  request { Ping }
}

componentdef ParentC {
  component PingerC() pinger;
  component PongerC() ponger;

  !connect ponger => pinger: PingPongPort;
}

componentdef PingerC {
  requires PingPongPort ppp;

  handle startHandler => control : Start e {
    !trigger Ping.event => ppp;
  }

  handle pongHandler => ppp : Pong pong {
    System.out.println("Got Pong");
  }
}

componentdef PongerC {
  provides PingPongPort ppp;

  handle pingHandler => ppp : Ping ping {
    System.out.println("Got Ping");
    !trigger ping => ppp;
  }
}

public class Main {
  public static void main(String[] args) {
    Kompics.createAndStart(ParentC.class, Init.NONE);
  }
}
```