A DSL for Distributed, Reactive Workflows

Mathijs Saey
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
mathsaey@vub.ac.be

Abstract
A typical large-scale, data driven, processing pipeline consists of a great amount of generic components that may be reusable inside other applications. However, there is currently no programming model that makes it possible to reuse existing data driven software components in a cluster-based setting. Offering such a model would promote the reuse of existing software, saving development time. Our work introduces the notion of reactive workflows, a technique that combines concepts from scientific workflows and reactive programming. We investigate the required abstractions to create such a workflow and offer these in a domain specific language, called Skitter. We plan to evaluate this framework based on its expressivity and horizontal scalability.

CCS Concepts • Software and its engineering → Domain specific languages; Distributed programming languages; Data flow languages;

Keywords Scientific Workflows, Distributed Programming, Reactive Programming

1 Motivation
The ubiquity of smartphones and the advent of the “Internet of things” made it possible for companies to access an enormous amount of real-time data. The heterogeneity of this data implies that there is often a need for preprocessing as data needs to go through (several) cleaning and transformation steps before it can be used [5]. Examples of such steps include the removal of duplicate or unnecessary data, changing the data format and filtering out unusable data points. Similarly, data is often archived to ensure its availability for future use. Embedding the logic of these preprocessing and archival steps in Big Data applications is a tedious process, and prevents the reuse of these steps in other applications that operate on similar data streams in real time.

We aim to enable the reuse of various data processing steps. This would lead to the following key advantages: first, common steps, such as the aforementioned preprocessing and archival, can be separated from application-specific logic; this makes it easier to reason about the individual data processing steps. Second, steps can be reused and shared across different applications; to make this possible we explicitly target the reuse of existing data (pre)processing software. Third and finally, separated data processing steps can be offered as a set of components, that can be distributed over a cluster, facilitating scalable, real-time data processing.

2 Problem
Our research aims to identify the correct techniques to compose a set of existing data processing steps in such a way that they can be efficiently executed on a cluster machine. The real-time nature of our problem domain implies that these processing steps should be able to accept data as soon as it enters the system from the outside world. Composing existing data processing steps in such a way that they can be executed on a cluster is not trivial, as programs that are executed on a cluster need to reason about partial failure and replication concerns. Concretely, we aim to tackle the following problems:

I. What new language abstractions are required to provide a runtime system with the necessary meta information in order to encapsulate, replicate and distribute existing software components?
II. What are the fundamental set of operators required to glue together these various components in one distributed resilient workflow?
III. How to accommodate nonfunctional concerns such as fault tolerance and performance on the language level as well as within the execution engine?

Current approaches for large-scale or data driven processing, such as stream processing [7], scientific workflows [6]
or reactive programming [3] do address some of these problems. However, to our knowledge there is no approach yet that focuses on all three problems.

**Stream processing** Throughout the last decade, there has been a great deal of work that focuses on large-scale data processing (commonly known as Big Data processing). More recently, some of this work focuses on processing real-time data with stream processing frameworks. These frameworks are highly relevant to our problem area, and are designed with distributed execution in mind. Unfortunately, these frameworks achieve scaling by placing limitations on the computations that can be expressed, which poses issues when attempting to reuse existing data processing software.

**Scientific workflows** Previous work makes it possible to combine existing data processing logic into scientific workflows; however, none of these systems are data driven, i.e. they don’t automatically start to process data when it arrives from some external data source (such as a cellphone or a sensor). Instead, these tools are almost entirely query-driven. A consequence of this query-driven nature is that scientific workflow systems are all inherently batch-based: they work on a complete data set, which is not compatible with the real-time nature of the applications we target.

**Reactive programming** Work in this area lead to the design of programming languages that automatically respond to data as soon as it arrives from an external data source. However, there are very few reactive languages that automatically distribute their computations over a cluster. Furthermore, reactive languages have issues dealing with long lasting computations and effectful statements [8], which are required by contemporary data processing applications.

### 3 Approach

We aim to solve these problems by means of a new domain specific language called Skitter. Skitter is centered around the concept of reactive workflows, which combines notions from the fields of reactive programming, scientific workflows and, to a lesser extent, stream processing. Reactive workflows consist of a set of connected reactive components. These components represent a single data processing step that is automatically executed when data enters the reactive workflow, or when a connected component produces new data. Our language consists of two parts: the textual definition of reactive components and the visual composition of these components into a reactive workflow. We envision that the reactive components are written by developers with experience in reactive programming. The workflows can be designed visually by domain experts, who may not have any programming experience. Skitter’s underlying runtime can then distribute the execution of a reactive workflow over a cluster. We implemented Skitter on top of the Elixir\(^1\) programming language. We decided on Elixir for its focus on distributed systems and because it runs on top of the Erlang VM, which has a proven track record of scaling to large systems (used by Amazon, Facebook, Ericsson, . . . ). Furthermore, Elixir implements the actor model [1] which is a natural fit for reactive components.

**Component definition** Skitter’s component definition language aims to find the correct abstractions for the expression of generic, reusable data processing components (problem I). A simple example of such a definition can be found in Listing 1; this example computes the running average of any values it receives. A component definition consists of a set of functions that manage the lifetime of a component instance (e.g. the \texttt{init} function on line 5) and which allow it to \texttt{react} to incoming data (\texttt{react}). In the body of these functions, the developer can access the (potentially mutable) state of the component instance. Skitter’s runtime ensures that this state is correctly distributed and replicated and that it can be recovered in the case of failure. It is also possible to write wrappers around existing data processing software. Existing software may trigger side-effects (e.g. I/O) which cannot be tracked by Skitter. Therefore, we introduce the notion of \texttt{effects} (shown on line 3 in Listing 1), which enable a component developer to provide additional information about the side-effects a component may trigger when it reacts to incoming data. The runtime uses this information to automatically handle partial failure and replication concerns.

**Workflow definition** Skitter’s workflow language makes it possible to compose the components we just defined (problem II). We plan to offer our workflow language as a visual programming language; this is done to enable non-experts to create workflows out of existing components. Developers can create a workflow by connecting the output of a component to the inputs of other components; an example of such a workflow can be seen in Figure 1. In this example, we aim to visualize a metric (e.g. the noise level) of a specific area in a city. All the noise and coordination data enters

```plaintext
Listing 1. A Skitter component that calculates the average of all the numbers it receives.

```n
c
```

\footnote{https://elixir-lang.org/}

```
We plan to evaluate our expressivity hypothesis by implementing a number of existing data processing components in Skitter, and composing these into a workflow. The majority of data processing components we plan to implement will be sourced from a citizen science project that is currently active at our research group. We will evaluate the component implementations based on their simplicity compared to the original implementation. These two versions can then be evaluated, for instance, by comparing how much code is required to implement various concerns (e.g. distribution, fault-tolerance, …). Besides this, we will write wrappers for existing data processing software and examine if our effect system is expressive enough to correctly handle the partial failure and replication concerns for these programs. To evaluate the ease-of-use and expressiveness of Skitter’s workflow language we will implement various use cases of the aforementioned project. Finally, we will evaluate the horizontal scalability of our approach by benchmarking various use cases on a cluster machine. We can consider both artificial and real-world benchmarks for this evaluation. The artificial benchmarks will be selected to investigate the asymptotic behavior of our system, while the real-world benchmarks will be sourced from the aforementioned project.

Acknowledgments
This work is funded by the Flamenco project of the Flemish agency for Innovation by Science and Technology.

References

Figure 1. A Skitter workflow that calculates the average noise level within a certain geographical area.

the Geographical Filter, which filters the data within the desired area. All the measurements inside the area are persisted by the Data Storage component and we compute an average using the Geographical Average component, which is visualized in a live view.

Workflow execution We are currently developing an execution engine which efficiently executes a reactive workflow on a cluster (problem III). This problem can be divided into two subproblems: workflow execution and component orchestration. To address workflow execution, our engine attempts to extract the maximum amount of parallelism that is present in a given workflow. To this end, our engine is based on the dataflow model [4]. In this model, an operation (i.e. an invocation of react) can be executed when all of its inputs are present. We decided to use this model due to its ability to extract the latent parallelism present in any program, as multiple operations are allowed to execute in parallel, if this is permitted by the data dependencies. For instance, in our example workflow the “Data Storage” and “Average” component instances can react to incoming data in parallel. The data-driven nature of the applications we target makes it possible to process various data records entering the system in parallel. To achieve this, our execution strategy relies on a variant of the dataflow model, called tag-token dataflow [2]. As the name implies, in this model, a tag is added to each token (i.e. a data record), that allows the runtime to differentiate between tokens from different execution contexts. Our workflow execution strategy uses these tags to process different data records concurrently. Since our execution strategy effectively isolates the execution of each data processing step, our runtime includes the use of a component orchestration algorithm. This algorithm ensures that the state of the various replicas of a component remains consistent; it is also responsible for ensuring that this state can be recovered if failure occurs. To do so, this algorithm uses the information provided by the effects of a component.

4 Evaluation
We aim to test two hypotheses. Our first hypothesis deals with expressivity: we claim that Skitter is sufficiently expressive to implement a variety of different reactive components. It should be possible to define new components, as well as easily reuse existing data-driven processing logic. Our second hypothesis deals with scalability: we claim that our dataflow based workflow execution strategy is horizontally scalable (i.e. it can scale with additional computational resources).