White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of York.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/164779/

Version: Accepted Version

Proceedings Paper:

Kolovos, Dimitris, Neubauer, Patrick, Barmpis, Konstantinos et al. (2019) Crossflow: A framework for distributed mining of software repositories. In: Proceedings - 2019 IEEE/ACM 16th International Conference on Mining Software Repositories, MSR 2019. 16th IEEE/ACM International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019 IEEE International Working Conference on Mining Software Repositories. IEEE Computer Society, CAN, pp. 155-159.

https://doi.org/10.1109/MSR.2019.00032

Reuse

Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS

University of Sheffield

UNIVERSITY of York

CROSSFLOW: A Framework for Distributed Mining of Software Repositories

Dimitris Kolovos*, Patrick Neubauer*, Konstantinos Barmpis*, Nicholas Matragkas*, Richard Paige*†
†Department of Computing and Software, McMaster University, Canada
*Department of Computer Science, University of York, United Kingdom
{firstname.lastname}@york.ac.uk

***Abstract*—Large-scale software repository mining typically requires substantial storage and computational resources, and often involves a large number of calls to (rate-limited) APIs such as those of GitHub and StackOverflow. This creates a growing need for distributed execution of repository mining programs to which remote collaborators can contribute computational and storage resources, as well as API quotas (ideally without sharing API access tokens or credentials). In this paper we introduce CROSSFLOW, a novel framework for building distributed repository mining programs. We demonstrate how CROSSFLOW can delegate mining jobs to remote workers and cache their results, and how workers can implement advanced behaviour such as load balancing and rejecting jobs they cannot perform (e.g. due to lack of space, credentials for a specific API).**

***Index Terms*—Open source software, Public domain software, Data collection, Data integration, Data analysis, Pipeline processing, Data flow computing, Software engineering, Client-server systems, Computer aided software engineering, Modeling, Scalability, Distributed processing.**

I. INTRODUCTION

In [1], we set out to assess the "popularity" of 22 different model-driven engineering technologies by measuring their use in open-source GitHub repositories. To achieve this we started by querying GitHub for files with relevant extensions and keywords. To expand our search and retrieve as many files as possible, we collected all repositories of the files returned by the first round of searches (1,900+ repositories overall), and we then queried every repository again (through the GitHub API) for files of all technologies of interest. This step alone required $22 \times 1,900$ GitHub API calls, which vastly exceeded the 5,000 calls rate limit per hour imposed by GitHub. Given that our team comprised several collaborators, one way around this limit would have been for each collaborator to generate a GitHub OAuth personal access token and to then collect these tokens in one machine and rotate among them. As not everyone felt comfortable with this option, we decided to use one GitHub account and make our data collection program wait for the API rate limit to be replenished, which led to an overall execution time of more than 8 hours.

This experience motivated us to investigate approaches for distributed execution of software repository mining programs that would allow remote collaborators to contribute their API call allowances and computational resources, without having to share credentials or pre-authorised OAuth keys. This work resulted in the development of CROSSFLOW, a Java-based framework for development and distributed execution of multi-step software repository mining programs (workflows). Before resorting to implementing a new distributed execution framework, we attempted to build on top of existing frameworks such as Apache Spark and Flink, but they were not able to accommodate our locality scheduling requirements, discussed in Section II-F, and do not provide sufficiently fine-grained job-result caching facilities.

The rest of the paper is organised as follows. Section II, presents the architecture of CROSSFLOW, the domain-specific language it uses for specifying software repository mining workflows, as well as key features such as caching and locality scheduling. Section III presents related work and section IV concludes the paper and discusses directions for future work.

II. CROSSFLOW

CROSSFLOW is a novel distributed data processing framework tailored to the needs of collaborative repository mining. CROSSFLOW supports distributing tasks of multi-step repository mining programs, which we call *workflows* in the remainder of the paper, over multiple computing nodes (workers), which communicate through, and are orchestrated by, a master node using messaging middleware (Apache ActiveMQ [2] in our current implementation). We explain the building blocks and facilities of CROSSFLOW through a running example.

In this example we wish to discover the degree to which different technologies (e.g. programming languages, tools) are used together in the same GitHub repositories. For instance, we wish to discover if projects using the Eclipse Graphical Modelling Framework (GMF)[1] are more likely to also use the ATL [3] or the QVTo[2] model transformation languages. One way to achieve this is to:

- Record information about the technologies of interest in a structured format. For example, the CSV file (technologies.csv) in Table I, captures a known file extension and keyword for each technology of interest;
- Query GitHub to find repositories containing files of interest for each technology (the GitHub search API returns the details of up to 1000 files for each search);
- Clone the repositories of all collected files and search the local clones for files of all technologies of interest;
- Compute a co-occurrence matrix like the one in Table II.

[1]https://www.eclipse.org/gmf-tooling/
[2]https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

Technology	Extension	Keyword
GMF	.gmfgraph	figure
ATL	.atl	rule
QVTo	.qvto	transformation

TABLE I: Technologies with file extensions and keywords

	GMF	ATL	QVTo
GMF		16	25
ATL	16		7
QVTo	25	7	

TABLE II: Technology co-occurrence matrix

A. Architecture

As illustrated in Figure 1, CROSSFLOW provides a purpose-built domain-specific language for modelling repository mining workflows, and a code generator that produces implementation scaffolding in Java, which depends on a reusable runtime library. While a CROSSFLOW model specifies the sources, tasks, streams and sinks of a workflow, how they are wired, and where tasks are executed (in all workers vs. only in the master node) it does not capture the behaviour of the modelled sources, tasks and sinks. This is expressed using hand-written Java code embedded in the generated scaffolding. Once the desirable behaviour has been implemented, the compiled workflow-specific code and the reusable core runtime library are bundled in a self-contained runable JAR file, which is executed on the nodes participating in the execution of the workflow. The following sections discuss the components of the CROSSFLOW architecture in detail.

B. CROSSFLOW *DSL*

The CROSSFLOW DSL has been implemented on top of the Eclipse Modelling Framework and its abstract syntax (metamodel) is illustrated in Figure 2. We explain its building blocks using a model (cf. Figure 3) of our running example.

Sources feed the workflow with jobs based on user input. In our example, *TechnologySource* reads a comma-separated file structured like Table I, producing *Technology* jobs, consisting

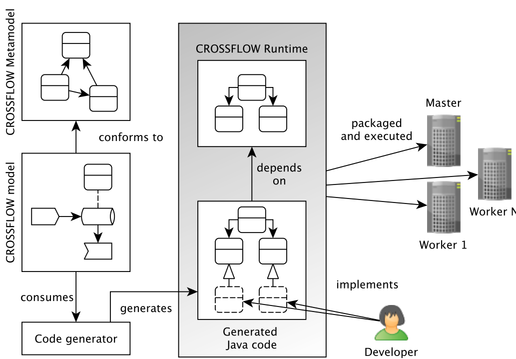

Fig. 1: CROSSFLOW Architecture

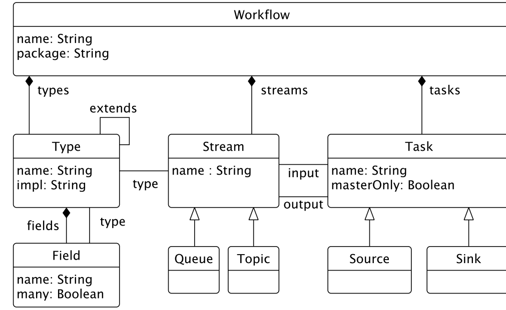

Fig. 2: CROSSFLOW Language Metamodel

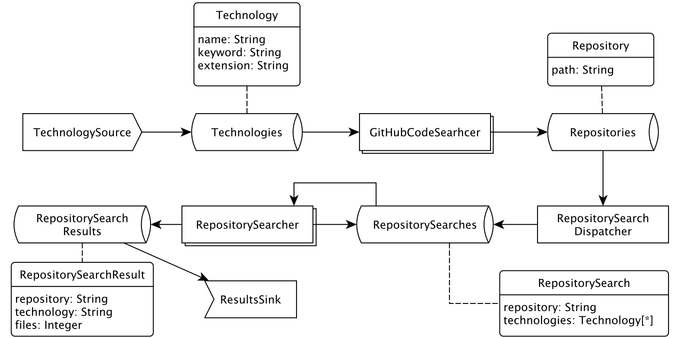

Fig. 3: CROSSFLOW Model of the Running Example

of a name, an extension and a keyword, into the *Technologies* stream. Sources are only executed on the master node.

Streams are message channels to which sources and tasks of the workflow can send jobs that other tasks can perform, or results that sinks can aggregate and persist. In CROSSFLOW, job steams are typed: for example the *Technologies* stream only accepts jobs of type *Technology* (dashed line in the diagram). CROSSFLOW supports two types of streams: queues which send each job to only one of the subscribed workers, and topics which send each job to all subscribed workers.

Tasks subscribe to one or more (incoming) streams and receive jobs posted there by other tasks or sources. They can also post new jobs to one or more outgoing streams. For example:

- the *GitHubCodeSearcher* task subscribes to the *Technologies* stream, processes incoming jobs of type *Technology* by searching for files with the specified keyword and extension through the GitHub API, and for each result, it pushes its repository path (wrapped into a *Repository* job) to the *Repositories* stream. In a distributed execution of this workflow, each worker contributes an instance of *GitHub-CodeSearcher* (hence the double rectangle node shape in the diagram) which can perform such GitHub searches under its own credentials (and more importantly, its own rate limit);
- the *RepositorySearchDispatcher* task receives these repositories, and for every repository it has not encountered before, it produces one *RepositorySearch* job into the *RepositorySearchesStream*. Unlike *GitHubCodeSearcher*, *RepositorySearchDispatcher* is represented with a single rectangle, signifying that only one instance of the task is executed and this instance lives on the master node. This does not have

a significant impact on the performance of the workflow as the cost of filtering out duplicate repository IDs is negligible to that of querying GitHub and cloning Git repositories;
- for each *RepositorySearch* job, the *RepositorySearcher* makes a shallow clone of the Git repository and counts the number of files with the extension and containing the keyword relevant to each technology.

Sink components can subscribe to streams and receive results to aggregate/persist. For example, the *ResultsSink* sink in the example, collects *RepositorySearchResult*s, builds the co-occurrence matrix illustrated in Table II, and periodically persists it into another CSV file (results.csv). As with sources, sinks are only executed on the master node.

C. Code Generator

The CROSSFLOW code generator can consume a workflow model and produce strongly-typed scaffolding Java code. In particular:

- For every *Task* and *Sink* in the model, it produces an abstract base class, as well as a skeleton subclass which contains one *consumeXYZ(...)* method for every incoming stream, where hand-written code needs to be added to handle incoming jobs. For example, from the *GitHubCodeSearcher* task of Figure 3 the generator produces an abstract *GitHubCodeSearcherBase* class as well as a concrete *GitHubCodeSearcher* class that extends the base class and contains an empty implementation of a *void consumeTechnologies(Technology technology)* method, which is called by the workflow when a new *Technology* job is received. The hand-written body of the method is illustrated in Listing 1.
- Similarly, for every *Source* in the model, the generator produces an abstract base class with infrastructure-communicating code and an abstract *void produce()* method, as well as a concrete sub-class for developers to implement the latter and specify the behaviour of the source (e.g. in the case of *TechnologySource*, the implementation of *produce* reads file extensions and keywords from an input CSV file and pushes *Technology* jobs to the *Technologies* stream)
- The abstract base classes generated from *Task*s and *Source*s, also contain one *sendXYZ(...)* method for each outgoing stream that developers can use to send outgoing jobs to the respective stream. For example, *GitHubCodeSearcherBase* contains a *void sendToRepositories(Repository repository)* that is used by hand-written code in its concrete subclass to send *Repository* jobs to the *Repositories* stream, for *RepositorySearchDispatcher* to consume.
- For every *Stream* in the model, the generator produces a Java class which contains code that subscribes instances of the concrete task classes to the underlying ActiveMQ topics/queues [2]. Unlike with tasks, sources and sinks, developers do not need to write additional code to specify the behaviour of streams.
- For every *Type* in the model that doesn't specify an *implClass* (i.e. is not a proxy for an existing Java class), the generator produces a Java class which contains properties, setters and getters for the type's fields. If there is at least one stream typed after the type in question, the generated Java class is made to extend the built-in *Job* class, which provides ID/correlation ID fields as well as serialisation capabilities which are required for caching as discussed below.

```
1  @Override
2  public void consumeTechnologies(Technology t)
       throws Exception {
3    List<String> paths = ...; // runs GitHub search
4    for (String path : paths) {
5      Repository r = new Repository();
6      r.path = path;
7      r.corellationId = t.id;
8      sendToRepositories(r); } }
```

Listing 1: The consumeTechnologies(...) hand-written method of GitHubCodeSearcher

The generator also produces a main Java class named after the workflow instance in the model, which starts and coordinates the execution of the workflow on a node, supporting command-line parameters through which users can specify:

- Whether the node runs in *master* or *worker* mode. Each workflow execution can be coordinated by one master node. In the master mode, additional command line parameters specify whether the workflow needs to start an embedded (ActiveMQ) messaging broker that will manage the message channels of the workflow, or can use an existing one at a specified IP address.
- For nodes in worker mode, relevant parameters can define the IP address where the ActiveMQ broker instance is running. An additional parameter (*-exclude*) can be used to exclude particular tasks from their execution by the worker. For example, a worker may exclude *GitHubCodeSearcher* from its execution if it doesn't have GitHub credentials, and contribute to the workflow through cloning and searching repositories by means of the *RepositorySearcher* task.

D. Worker Error Handling

Exceptions produced during the execution of hand-written code in *consumeXYZ(...)*/*produce(...)* methods[3], such as *consumeTechnologies(...)*, in workers are caught by the generated base classes and sent to a dedicated stream (*InternalExceptions*) together with the job that caused them. In the current version of CROSSFLOW, jobs that cause exceptions during their execution are not rescheduled, they remain in the *InternalExceptions* stream for inspection by developers. Temporary loss of network connectivity between the master and the worker nodes is handled through the message persistence and wait-and-retry capabilities of the supporting (ActiveMQ) messaging middleware.

E. Caching

Jobs performed in the context of a repository mining workflow can require fetching large volumes of remote data (e.g. cloning Git repositories) or making calls to rate-limited APIs. To avoid repeated execution of such jobs, CROSSFLOW provides built-in support for job-level caching. In CROSSFLOW, each job has a unique (auto-generated) ID and an optional correlation ID which records the ID of the job of which it

[3]The generated signatures of such methods allow exceptions to be thrown during their execution.

is an output. For example, Listing 1 shows a redacted version of the implementation of the *consumeTechnologies(...)* method of the *GitHubCodeSearcher* class, where outgoing *Repository* jobs are associated to the incoming *Technology* by setting the correlation ID of the former to the ID of the latter (line 7).

The master node intercepts jobs submitted to all streams and caches outputs against their respective inputs based on IDs and correlation IDs so that previously-seen jobs are not re-executed in subsequent runs of the workflow, but instead their cached outputs are re-used.

F. Locality Scheduling

The first time the example workflow is executed in a distributed setup, different worker nodes will end up with different cloned Git repositories as a result of the execution of their *RepositorySearcher* tasks. The next time the workflow is executed (e.g. after a bug fix or after adding more technologies in the technologies.csv input), *RepositorySearch* jobs should ideally be routed to nodes that already have clones of relevant repositories from the previous execution to avoid unnecessary cloning of the same repositories in different nodes.

To achieve this, we originally considered delegating the required book-keeping to the master node. In this approach, the master node would be responsible for "remembering" how Git repositories (and other expensive to re-fetch/compute resources) were distributed between workers. However, given that workers can appear/disappear at any point during the execution of the workflow, we opted for a simpler and more powerful approach which eliminates the need for centralised book-keeping by enabling CROSSFLOW worker tasks to reject jobs allocated to them. Using this feature, we can achieve the desired locality scheduling in *RepositorySearcher* as follows:

- *RepositorySearcher* receives a *Repository* to analyse
- If the worker has a clone of the repository in question, it accepts and performs the job
- If it does not have a clone of the repository:
  - If it is the first time the worker encounters this job it adds the ID of the job to a list of encountered jobs and rejects the job
  - If the ID of the job is already in the worker's encountered job list upon reception, it assumes that all other nodes have previously rejected the job, and accepts it

The main advantages of this approach is that it eliminates the need for book-keeping at the master node and that it allows worker nodes to dynamically reject jobs, which is useful in several scenarios (e.g. when a worker runs out of GitHub API calls or out of space in its local filesystem). On the flip side, it incurs a runtime overhead as in the first execution of the workflow above all *RepositorySearch* jobs will be rejected once (i.e. sent back to the master node) by each worker before they start getting accepted. This can become an issue in cases where job messages carry a lot of data so developers of CROSSFLOW programs are encouraged to keep such messages small and provide pointers to larger data as opposed to embedding it when possible (e.g. the path of a file on GitHub as opposed to its contents). In terms of fair allocation of work across the workers, this is delegated to the respective facilities of the ActiveMQ messaging middleware (round-robin message distribution). Preliminary experiments have provided no evidence of unfair allocation but this is an area for additional investigation. The described

III. RELATED WORK

Boa [4] is a domain-specific language and infrastructure for mining software repositories. Boa's infrastructure leverages distributed computing techniques to execute queries against a multitude of software projects. The Boa language enables the specification of analysis of Git repositories, but it does not allow the specification of more complex workflows, such as retrieving and combining data from two different sources.

Another tool for distributed mining of software repositories is King Arthur[4], which is part of the GrimoireLab[5] tool chain. King Arthur is a distributed job queue platform that schedules and executes data retrieval jobs from software repositories using Perceval [5], a dedicated Python library. This platform enables the orchestration and distribution of data retrieval jobs only, while CROSSFLOW enables the distribution of mining workflows. Moreover, CROSSFLOW workers can selectively choose jobs to undertake depending on their capabilities. This is not the case with King Arthur workers, which simply pick the next job from a queue whenever they are idle.

Finally, Boinc [6] is a software system that facilitates the creation and execution of public-resource computing projects and therefore it can support the execution of mining workflows. Boinc shares many similarities with CROSSFLOW. Namely, it supports distributed computation, workers are assigned jobs based on their computational capabilities, and locality scheduling is used. At the same time though, CROSSFLOW offers particular features that make it more suitable for repository mining workflows. First, although Boinc supports selective execution of jobs from workers, it is the server, which decides on the distribution of jobs based on their estimate of computational requirements. On the other hand, in CROSSFLOW workers choose their jobs as the master node is completely unaware of the exact composition of the system. This results to increased robustness to specific faults, such as as network and time-out errors. Moreover, Boinc does not provide any high-level, declarative way to specify workflows.

IV. CONCLUSIONS AND FUTURE WORK

This paper introduced CROSSFLOW, a novel framework for development and distributed execution of multi-step repository mining programs. CROSSFLOW provides a domain-specific language for designing distributed workflows as well as a code-generator that produces implementation scaffolding for developers to complement with hand-written Java code. CROSSFLOW uses asynchronous message-based communication and provides built-in support for job-level caching and locality scheduling.

We are currently working on the implementation of concrete workflows for the needs of our industry partners in the collaborative project supporting the development of CROSSFLOW, and extending the framework with new facilities in the process.

[4]https://github.com/chaoss/grimoirelab-kingarthur
[5]https://chaoss.github.io/grimoirelab/

References

[1] D. S. Kolovos, N. D. Matragkas, I. Korkontzelos, S. Ananiadou, and R. F. Paige, "Assessing the use of eclipse mde technologies in open-source software projects." in *OSS4MDE@ MoDELS*, 2015, pp. 20–29.

[2] B. Snyder, D. Bosnanac, and R. Davies, *ActiveMQ in action*. Manning Greenwich Conn., 2011, vol. 47.

[3] Frédéric Jouault and Ivan Kurtev, "Transforming Models with the ATL," in *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, ser. LNCS, Jean-Michel Bruel, Ed., vol. 3844, Montego Bay, Jamaica, October 2005, pp. 128–138.

[4] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 422–431. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606588

[5] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, "Perceval: software project data at your will," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 1–4.

[6] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE, 2004, pp. 4–10.

[7] A. Bagnato, K. Barmpis, N. Bessis, L. A. Cabrera-Diego, J. D. Rocco, D. D. Ruscio, T. Gergely, S. Hansen, D. S. Kolovos, P. Krief, I. Korkontzelos, S. Laurière, J. M. L. de la Fuente, P. Maló, R. F. Paige, D. Spinellis, C. Thomas, and J. J. Vinju, "Developer-Centric Knowledge Mining from Large Open-Source Software Repositories (CROSSMINER)," in *Software Technologies: Applications and Foundations - STAF 2017 Collocated Workshops, Marburg, Germany, July 17-21, 2017, Revised Selected Papers*, 2017, pp. 375–384. [Online]. Available: https://doi.org/10.1007/978-3-319-74730-9_33

[8] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Eclipse development tools for epsilon," in *Eclipse Summit Europe, Eclipse Modeling Symposium*, vol. 20062, 2006, p. 200.

[9] S. Madani and D. S. Kolovos, "Re-Implementing Apache Thrift using Model-Driven Engineering Technologies: An Experience Report," in *Proceedings of the 16th International Workshop on OCL and Textual Modelling co-located with 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 2, 2016.*, 2016, pp. 149–156. [Online]. Available: http://ceur-ws.org/Vol-1756/paper11.pdf

[10] D. S. Kolovos and R. F. Paige, "Towards a modular and flexible human-usable textual syntax for EMF models," in *Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018.*, 2018, pp. 223–232.

Appendix

The source code of the tool is currently available on a public repository on GitHub[6]. The tool is developed in the context of a large collaborative research project, Crossminer, involving several industrial and academic partners [7]. All of them have a vested interest in continuing the development and maintenance of the tool in the future, and thus the tool will remain available in the future on GitHub. Currently, the tool is build on Eclipse Epsilon [8], Apache Tomcat, Apache ActiveMQ [2], and Apache Thrift [9].

There are two use cases for the tool: a new mining workflow is specified and then executed; an existing workflow is executed. Detailed instructions on how to execute the tool are provided in the repository. A brief version is provided below.

Specify new workflow: To specify a new workflow, a user needs an Eclipse Epsilon distribution[7], Apache Tomcat[8], ActiveMQ[9] and Thrift[10]. Instructions on how to install these are provided on our repository. Once these dependencies are installed, the user will need to clone our repository and import the projects in Eclipse. Our tool uses Apache Ivy[11] for dependency management, so once the projects are imported to Eclipse all dependencies should be resolved automatically. To specify the new workflow, the user has to create a new Eclipse project and specify the workflow in an XML-based, human-friendly notation called Flexmi [10]. Once the workflow is specified, the code generator can be invoked to generate the code skeleton. Developers then need to extend the generated source, task and sink classes with the desirable behaviour. Once the code is implemented and compiled, it needs to be bundled into a runnable JAR which can then be executed as a standard Java application on the master and worker nodes.

Execute workflow: To execute a workflow, the user simply has to execute the runnable JAR produced in the previous step on the command line. The JAR has to be executed on every machine, which wants to contribute computational resources to the analysis of the workflow. The user can specify the name of the workflow, the type of the node (master or worker), the IP address of the master node, and the listening port of the master node by means of the parameters *name*, *mode*, *master*, and *port*, respectively.

The current repository of the tool comes with detailed instructions on how to perform the above process[12]. Moreover, the repository contains predefined workflows that users can execute in order to experiment with the platform. The repository contains the model and the code for the examples, as well as how to generate JARs in order to execute them.

[6] https://github.com/crossminer/scava/tree/crossflow/crossflow
[7] https://www.eclipse.org/epsilon/download/
[8] http://tomcat.apache.org
[9] http://activemq.apache.org
[10] https://thrift.apache.org
[11] http://ant.apache.org/ivy/
[12] https://github.com/crossminer/scava/tree/crossflow/crossflow/README.md