

High-Level Modeling and Low-Level Adaptation of Serverless Function Choreographies

Bachelor Thesis

Benjamin Walch

01518922

Supervisor: Shashko Ristov, PhD

Innsbruck, May 11, 2020

Declaration

By my own signature I declare that I produced this work as the sole author, working independently, and that I did not use any sources and aids other than those referenced in the text. All passages borrowed from external sources, verbatim or by content, are explicitly identified as such.

I consent to the archiving of the bachelor thesis at the institute / faculty:

Date

Signature

Abstract

The Distributed and Parallel Systems group developed the *Abstract Function Choreography Language* (AFCL). It is a specification for describing serverless workflows. Furthermore, a Java API, to describe serverless application workflows programmatically, was developed. The product which results in using that API is a workflow being described in AFCL in a text based file. Until now, the workflow being described in the text file has to be created manually (by editing the file directly), or a programmer has to write Java code which utilizes the API to generate the file.

The aim of this bachelor project is to develop a visual workflow editor, which makes modeling of workflows possible at a high level of abstraction. The tool should not only be able to load, display and save workflows, but also optimize workflows for multiple FaaS provider(s) in case of quotas and limits, and also in case of performance.

Contents

1	Introduction	1
2	Background	5
2.1	Serverless Computing	5
2.2	FaaS	6
2.3	Serverless Functions	6
2.4	Serverless Workflows	6
2.5	AFCL	7
2.5.1	Overview	8
2.5.2	Control-flow	8
2.5.3	Properties and Constraints	8
2.5.4	Data-flow	9
2.6	Tools and Technologies	9
2.6.1	npm	9
2.6.2	webpack	9
2.6.3	ECMAScript	10
2.6.4	Babel	10
2.6.5	SASS	10
3	AFCL ToolKit	11
3.1	System Architecture	11
3.1.1	Frontend	12
3.1.2	Backend	13
3.2	System Design	13
3.2.1	Frontend	14
3.2.2	Backend	23

system ar-
chitecture
and design?

4	Web Interface	29
4.1	Dashboard	29
4.2	Functions	30
4.3	Editor	31
4.3.1	Drawing Area	32
4.3.2	Sidebar	33
4.3.3	Property View	34
4.3.4	Toolbar	34
4.4	Settings	36
5	Evaluation	37
5.1	Methodology	37
5.2	Composing Workflows	38
5.3	Adapting Workflows	39
6	Conclusion	41
6.1	Future Work	41
A	AFCL API functions	43
	Bibliography	47

List of Figures

3.1	system architecture	12
3.2	frontend stack	14
3.3	mxCell model	17
3.4	A graph showing cells and their associated user objects	18
3.5	workflow adaptation	26
4.1	the dashboard	30
4.2	the function repository	31
4.3	the editor	32
4.4	graph modeling	33
4.5	the sidebar with a dropdown open	33
4.6	the toolbar	34
4.7	graph validation	35
5.1	Evaluated development effort for creating GCA, AD and GEN AFCL workflows using different methods. method (1): edit text file manually, method (2): AFCL Java API, method (3): AFCL ToolKit	38
5.2	Evaluated development effort for adapting workflows using different methods. method (1): edit text file manually, method (2): AFCL Java API, method (3): AFCL ToolKit	40

List of Tables

3.1 Implemented Cells, *: AFCL Object 19

1 Introduction

The abstract does not show why do we need a GUI to create workflow. I would say that domain experts who know the business, are usually not familiar with serverless technology, thereby they would have difficulties to develop the workflow with the API. Manual is slow, so we need some toolkit to facilitate workflow development... Somehow like this.

”Run code, not Servers” is a recent term in the cloud computing world. With the rise of serverless technologies during the last few years, *Function-as-a-Service* (FaaS) has become increasingly popular. This cloud computing concept offers new advantages for software development: very high flexibility, nearly unlimited scalability and pay-per-use pricing. Entire programs can be defined in workflows and can be executed using serverless technology in the cloud.

However, each serverless provider has its own definitions on how a workflow is expressed, as well as different pricing models, limits and quotas. For example, the maximum number of concurrent function invocations is limited to 1000 concurrent invocations for current FaaS providers at time of writing.¹ In addition, different providers also support different programming languages. That being said, a workflow from one system is not compatible with another. This means, the user is bound to a specific provider (vendor lock-in), if they want to reuse a workflow.

Many FaaS providers offer tools which should support and simplify the creation of workflows in their environment. Nevertheless, this is still a complex task. Most of the time, a software developer is needed, because others simply lack the necessary know-how; and even for developers, this task can be tedious and time-consuming.

The distributed and parallel systems group developed a specification to describe serverless workflows at a high abstraction level. This specification is called *Abstract Function*

¹checked providers: AWS, Google, IBM and Microsoft

Choreography Language (AFCL) and was created to overcome the weaknesses of current FaaS platforms - for example, the incompatibility of workflows between different providers - and makes cross-cloud workflow execution possible.

Castro et al. mention that, "One of the major challenges slowing the adoption of serverless is the lack of tools and frameworks" [2].

Therefore, the goal for this thesis is to develop a system, which facilitates the development of AFCL workflows and focuses on business logic.

write weaknesses of existing work here (visualization, without development options, which is still manual!), then after the sentence "This thesis ..." you will write our solution and how do you advance compared to state of the art.

In addition, visualizing - in particular loading, displaying and editing - of existing AFCL workflows in should be supported. The system should also provide an interface for performing workflow adaptations at low-level, based on specific inputs.

This thesis fulfills the above goals and overcomes the aforementioned problems. With the developed system, which we will refer to as the "AFCL ToolKit" in the text, below, it is possible for non-developers to create a workflow in less time than it would have taken a developer to write the workflow in AFCL.

Another advantage is that workflows created with AFCL ToolKit are guaranteed to be valid in AFCL, which minimizes the rate of human errors.

Regarding provider limitations, a scheduler (which is not part of this thesis) would decide to distribute for example one big concurrent loop - with more iterations than the maximum number of function invocations allowed by the provider - onto multiple FaaS providers. For this purpose, a low-level adaptation service is provided which offers an interface for specific workflow adaptations.

Why this service is important? Because AFCL has a limit to specify the location of each function. In order to allow a loop to be executed across multiple regions / providers, we need to unroll the loop and specify function locations within a loop. This is missing, which is very important why your work is important. Mention in provider limits in Section 2 that their languages are limited to run workflows within a single region.

I would mention concrete problem and concrete solution. First both problems of SOTA - visualization only, without development, which is manual in text file or in JS, which requires de-

It is assumed that concrete implementations of the functions are already developed and deployed to a FaaS provider.

Better name chapter numbers.

In the next chapter 2, we will give background information on topics related to this thesis. After that, in chapter 3, we will look closer at the implementation of AFCL ToolKit, in particular the system architecture and design, before the application's interface is presented and explained in chapter 4.

Finally, we will evaluate AFCL ToolKit and clarify its benefits to the developed system in chapter 5.

2 Background

This chapter contains important details on the terms and technologies used in this thesis and helps the reader to better understand the further topics.

First of all, the serverless concept, in particular Function-as-a-Service, Serverless Functions and Workflows are explained, before giving a brief overview about AFCL. The tools and technologies used for development are also presented to readers who are not yet familiar with them.

Can you switch the term into function choreographies? Define once in introduction and use the abbreviation FC instead of workflow.

2.1 Serverless Computing

Serverless computing is widely known as an event-driven cloud execution model. In this model, the client provides the code and the cloud provider manages the life-cycle of the execution environment of that code. The idea is based on reducing the life span of the program to execute functionality in response to an event. Hence, the program's processes are born when an event is triggered and are killed after the event is processed [5]. Developers can completely focus on the program they need to run and not worry at all about the machine it is on or the resources it requires [9]. Castro et al. define serverless computing as follows: "Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running" [2].

The term "serverless" can be misleading though, as there are still servers providing these backend services, but all of the server space and infrastructure concerns are handled by the vendor. Serverless means that the developers can do their work without having to worry about servers at all [4].

2.2 FaaS

FaaS is a form of serverless computing, which is disrupting the way applications and systems have been built for decades. By abstracting infrastructure provisioning and deployment, user-provided functions can be invoked and executed remotely. The user only has to worry about development and triggering of the function. This serverless runtime has not only the advantage that it avoids costly pre-allocated or dedicated hardware, but also offers almost unlimited possibilities in scalability. FaaS systems are designed to allow a usage-based billing, what means the user will only be charged for resources required during execution.

2.3 Serverless Functions

This

Serverless functions - or tasks - are single-purpose, mostly stateless, programmatic functions that are hosted on cloud infrastructure. These infrastructure is provided through a FaaS platform. They are event-driven and can be invoked through the internet, mainly using HTTP. Like conventional functions, they accept arguments and return the result of the computation - the difference being that this is done over a network.

2.4 Serverless Workflows

On an abstract level, a workflow consists of a set of interdependent tasks that need to be executed to achieve a specific goal. [...] A task within a workflow has the following properties: dependencies on the software or service used by the task to perform its computation (software flow), dependencies on data (data flow), and dependencies on other tasks (control flow) [3].

I would not mention other works than AFCL, because you do not support those patterns in your toolkit. Just citation [3] for global control and data flow.

Basic and advanced control flow patterns for workflows are shown in [8].

A serverless workflow is a complex workflow defined through the composition of serverless functions, connected by control- and data flow. Serverless workflows make it possible to combine and reuse serverless functions in order to build more complex applications. Workflows can declare the structure of applications - and using formats like YAML, XML or JSON, they can be described in a text-based file.

introduce here how Step, IBM and others create workflows and that the users are unable to port from one to another FC system.

2.5 AFCL

This section gives the reader a brief overview on AFCL and its features. More detailed information about AFCL as well as the API documentation can be found at [1].

The *Abstract Function Choreography Language* (AFCL) was created to overcome weaknesses in current FaaS platform implementations. As the name indicates, this specification describes serverless workflows at a high level of abstraction, forming a step on the way to cross-cloud workflow execution.

The following paragraph should be moved to the previous section 2.4. Here write only about positive issues of AFCL, because your work is based on it, so based on a good language :)

Let us consider a workflow built with AWS Step Functions, which should be executed on IBM Cloud. To make that work, the workflow has to be ported to the other platform to be compatible. In other words - it has to be recreated by a skilled programmer - for example by using the exported workflow as a template.

The AFCL specification also has more extensive control-flow and data-flow constructs available than current FaaS providers offer, in terms of language features. Therefore, AFCL can not only eliminate the incompatibility of workflows between different providers (vendor lock-in), but also opens the door to execute a workflow by distributing its computation over multiple FaaS providers.

To be able to actually execute such multi-cloud workflows, some further tools are needed: an Enactment Engine and a Scheduler.

2.5.1 Overview

AFCL is based on YAML¹ and ships with a schema². There exist two types of functions, base functions and compound functions, which can be connected by specifying control-flow and data-flow information. While base functions represent a single task, compound functions provide nesting - they can include some base functions or even other compound functions.

Base and compound functions have data input (`dataIns`) and data output (`dataOuts`) ports. Data input can refer to another data output or data input of another function, by specifying the name in combination with the data output port of the other function.

2.5.2 Control-flow

In AFCL, base or compound functions are specified one after another, which means that they are executed sequentially. However, the following control constructs are introduced with AFCL: `sequence`; `if-then-else`; `switch`; `for`; `while`; `parallel`; and `parallelFor`.

2.5.3 Properties and Constraints

Additional, optional attributes for `dataIns` and `dataOuts` ports of a function, or for the function itself can be defined in `properties` and `constraints`. While those are simple key-value pairs accepting string values, AFCL has a few defined properties and constraints to specify concrete attributes like invocation type of a function, element index or data distribution information in loops.

¹<https://yaml.org>

²<http://dps.uibk.ac.at/projects/afcl/files/schema/schema.yaml>

2.5.4 Data-flow

AFCL allows the expression of data-flow by connecting the source data ports of functions to the target data ports of functions. This offers support for more complex data-flow scenarios and might improve performance of the workflow. A source data port can be the input data port of the whole workflow, or a data port (input or output) of another function. So it is possible to even connect data ports of outer functions, with a lower nesting level, to inner functions, with a higher nesting level.

2.6 Tools and Technologies

2.6.1 npm

The *node package manager* is the world's largest software registry, where open-source software packages of developers and companies are shared all over the world.³ Over the last few years, *npm* has become a de-facto standard for package management in JavaScript development.

2.6.2 webpack

webpack is a module bundler, its main purpose is to bundle JavaScript code for the usage in a browser.⁴ In particular, multiple modules (often hundreds of them) with dependencies [to each other] are processed and bundled into a few files. To be able to process other types of files than JavaScript or JSON, webpack offers the opportunity to configure a **loader**. In this application, the following loaders are configured:

- babel-loader, to transform ES and React JSX to browser-compatible JavaScript
- sass-loader, to transform SASS to CSS
- css-loader, to transform CSS to CommonJS
- file-loader, to handle static resources like images and fonts

³<https://www.npmjs.org>

⁴<https://webpack.js.org>

2.6.3 ECMAScript

The scripting language specification *ECMAScript* (ES) was created to standardize JavaScript. With the release of ES6 (also known as ECMAScript 2015), features like class declarations, module imports and arrow function expressions became possible. After ES6, every year a new edition of the ECMAScript standard was finalized and released, offering new features. It is particularly worth highlighting here is the rest/spread operator feature which was released with ES9, as it was used a lot in this thesis.

2.6.4 Babel

Since current browsers only have partial support of ECMAScript, a *transcompiler* or *transpiler* is needed to transform the ECMAScript source code to JavaScript, which common browsers are capable of interpreting. *Babel* is an industry standard to transpile ES to common JavaScript or lower ES versions.⁵

2.6.5 SASS

CSS, in its pure form, reaches its limits when one thinks about using variables, functions or nested rules. *SASS* is a stylesheet language, which is⁶ - similar to ES - compiled to CSS and offers the aforementioned and other features. A lot of CSS Frameworks also provide their source code in SASS, often served with a large variable set which makes customization easy.

⁵<https://babeljs.io>

⁶<https://sass-lang.com>

3 AFCL ToolKit

system architecture and design?

This chapter gives the reader a detailed overview on the system architecture and the system design of AFCL Toolkit, including evaluations for technology decisions for specific tasks.

AFCL ToolKit should not only visualize AFCL Workflows, but also give non-developers the opportunity to create and edit workflows through a Graphical User Interface (GUI). This GUI ships with a powerful editor, including additional features like real-time validation, change history (versioning), clipboard (copy-paste) and automatic layouting. Furthermore, workflows can be adapted at low-level in order to divide large parallel loops - for example loops with more iterations than allowed by current providers - into several loops which run in parallel. By doing this, AFCL ToolKit also adapts all existing data flow to respect the new structure.

3.1 System Architecture

In general, this is too short section. Explain all white boxes, as well as communication with external stakeholders (user, scheduler, ...)

refer to Fig. 3.1. Use the same terminology in the figure. Web Frontend - AFCLToolkit Frontend, while instead of Java Backend - use AFCLToolkit Backend.

AFCL ToolKit is built on top of two sub systems which are operating mostly decoupled from each other. On the one hand there is the backend, responsible for low-level operations persistence of function data.

On the other hand is the frontend with the GUI, responsible for user interaction, data visualization and file I/O.

Those two sub systems interact with each other by sharing JSON-serialized objects through an API, which the backend provides and the frontend consumes.

mention access to the frontend is through a browser.

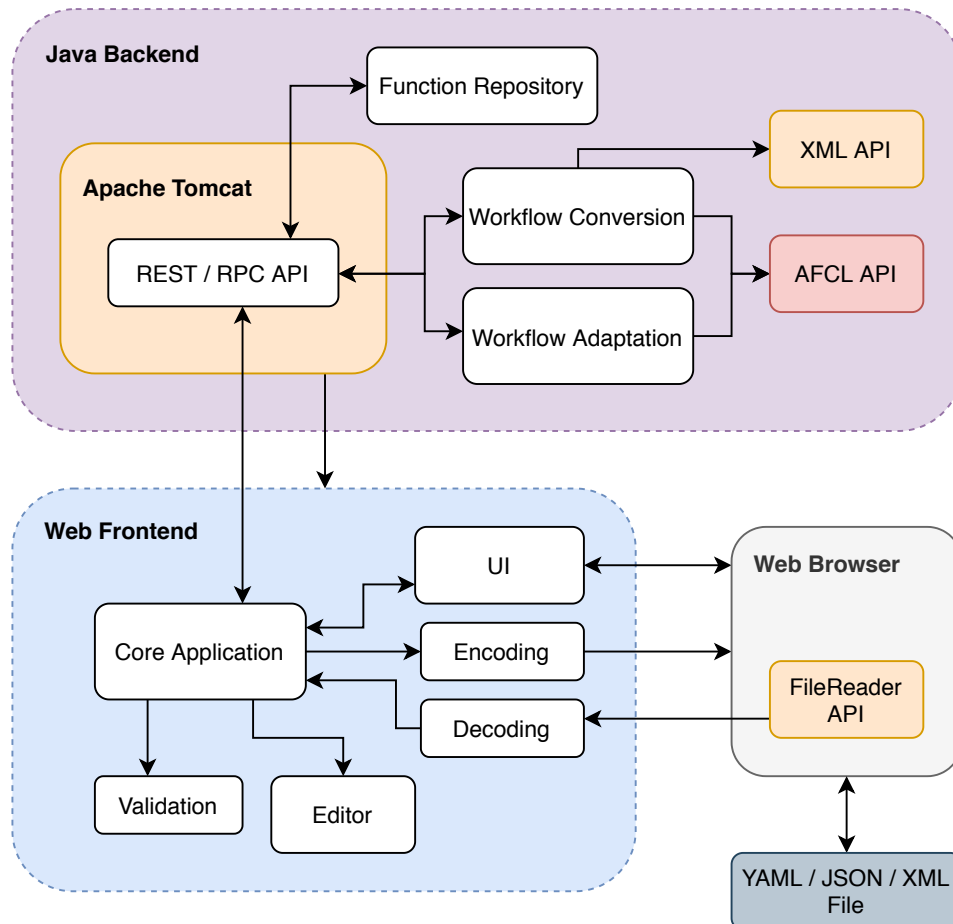


Figure 3.1: system architecture

3.1.1 Frontend

The frontend displays and controls the application's user interface. Its core component is the workflow editor, which is responsible for visualizing workflow data and provides a drawing area, allowing the user to compose and edit workflows in WYSIWYG manner.¹

¹<https://en.wikipedia.org/wiki/WYSIWYG>

can you please find some paper or link of report, rather than wikipedia?

The editor allows the user to load, create, edit and save AFCL workflows (YAML or JSON), which visualizes the current workflow as a graph. While editing, actions are constantly validated to a set of constraints which ensures workflows validity. The editor includes an action history which enables undo and redo, as well as a clipboard for copy and paste.

Where are load, export on the architecture? Encoding and decoding should be explained in the text, but user does load and store / export.

All features of the web-based user interface are described in detail in section 4.

3.1.2 Backend

The main purpose of the backend is to perform operations on workflows at low-level. These operations include the conversion of XML data delivered from the frontend, as well as complex workflow modifications. For the communication with the world outside, the backend includes a web server, which is used to expose the endpoint for its HTTP-based API to clients. Moreover, the frontend web application is also delivered by this web server. Last but not least, the persisting of function data is also covered by the backend.

3.2 System Design

My main concern about this section is that it tries to explain how did you do the programming, but not how did you solve the functionalities that are presented in Section 3.1. Let me give you an example. You explain the system architecture of a car and you say: I used tires from Goodyear, then wheel from China. Instead, you need to explain how the car moves left when you move the wheel to the left.

Beyond the overall system architecture, we will now look deeper into the system design. This section focuses on how the implementation of the system was achieved.

it seems that core application is the main part as it communicates with all other components. Maybe you do not need to present core app, but the editor? If the editor loads and stores, then shouldn't there be a link to the web browser?

no one knows at this stage about XML representation. Either you explain in the frontend, or use here intermediary representation. I prefer the former.

show this in the figure

isn't server a deploy-

3.2.1 Frontend

Can you synchronize white boxes of the architecture figure (frontend) with this section? Only core application is a subsection, but libraries, graph and others are not visible in the figure.

The frontend is built as a web application with HTML, CSS and JavaScript.

The base forms npm, which is used to manage and resolve dependencies. Webpack is the main part which orchestrates transpiling and bundling all sources and assets into single files by utilizing so-called loaders.

Babel is used as such a webpack loader for transpiling the ECMAScript and React JSX source code to browser-compatible JavaScript. The same applies for the CSS extension SASS, which is also integrated with a loader, and is used to make the process of styling the user interface more efficient.

webpack also executes some optimization plugins when building for production, which minimizes the size of the final bundle.

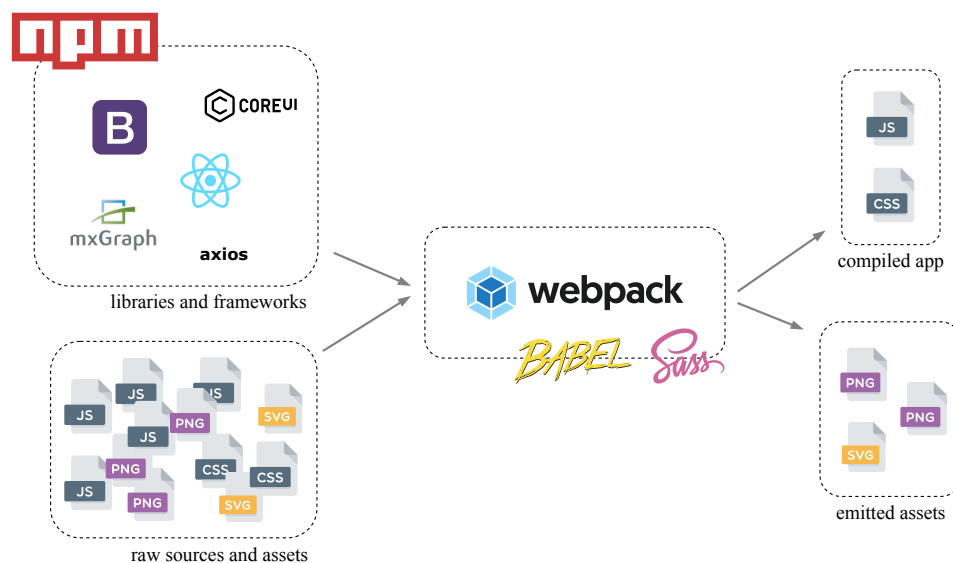


Figure 3.2: frontend stack

Libraries

Below is a list of selected libraries in this project. Care has been taken to ensure that all libraries used are open source.

These libraries should be included in background. Here you should write what is their advantage vs. others so that you used them.

- **React**

React is an JavaScript library for building user interfaces.²

- **mxGraph**

mxGraph is a JavaScript diagramming library.³

- **axios**

Axios is a JavaScript library providing a promise based HTTP client for networking tasks.⁴

- **Bootstrap**

Bootstrap is a toolkit for building web applications.⁵

- **CoreUI**

CoreUI is a free admin panel template, based on bootstrap.⁶

Core Application

The core application is divided into multiple modules, where each of them was designed to respect the single responsibility principle.⁷ A clean and nested structure of React components forms the user interface of the core application. One of the benefits when using React is that the data model displayed to the user typically maps very well to UI components.

system architecture and design?

²<https://reactjs.org>

³<https://jgraph.github.io/mxgraph/>

⁴<https://github.com/axios/axios/>

⁵<https://getbootstrap.com/>

⁶<https://coreui.io/>

⁷https://en.wikipedia.org/wiki/Single_responsibility_principle

Connection to the Backend

axios is used to communicate with the backend, by sending HTTP requests to the corresponding backend endpoint. Data which is sent in the HTTP body, is serialized using JSON, which JavaScript supports out-of-the-box.

Data model

All AFCL Java model classes have been ported to ECMAScript, to have all classes and properties available in the frontend, similarly to the backend. Although plain JavaScript objects would also do the job, this adds a kind of type-safety to the ECMAScript sources and improves readability of the code. Also, the data exchange with the backend and the encoding of the graph benefit from this approach, since the XML node names map to constructor names.

which ones?
This is im-
portant to
mention

Graph drawing

A workflow can be represented as a directed acyclic graph (DAG). DAG's are conventional models to present workflows, where nodes are tasks and edges are communications between tasks.

At the time of implementation, the JavaScript library *mxGraph* was the best choice for the graph visualization and modification.⁸ It has an outstanding documentation, enriched with a lot of demos and example code which show many use cases and extensive features, as well as a powerful production-grade example.⁹

The authors of *mxGraph* state:

“*mxGraph* is pretty much feature complete, production tested in many large enterprises and stable for many years. We actively fix bugs and add features [...]”

⁸see 3.2.1 to learn why

⁹<https://draw.io>

Graph model

In `mxGraph`, vertices and edges are represented by an `mxCell` model, which stores information about the position, dimension, geometry and style of the visual element. The `mxCell` class has been subclassed in a custom class `Cell`, in order to add user defined logic and an additional `type` property.

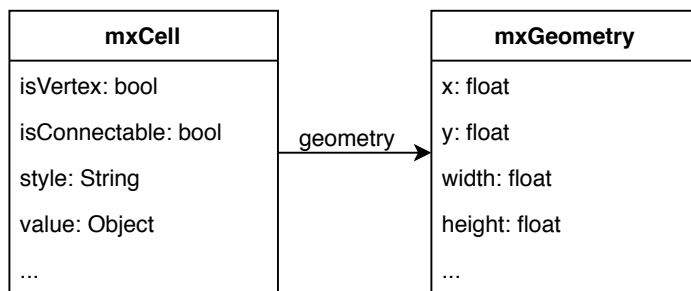


Figure 3.3: `mxCell` model

In the further text, the term *cell* refers to the visual element (which can be a vertex or an edge) in the graph, which represents always an instance of `Cell`, respectively.

Graph context

A *user object* can be associated with a `mxCell` through the `value` property. User objects give the graph its context, they store the business logic associated with the visual part. [6] For simple cases, the user object may be a string to display the cell's label. In more complex applications, the user object may be an object and some property of that object will generally be the label that the visual cell displays.

In this application, the cell's user objects are instances of the ported AFCL classes, and the function name of the AFCL object is used as the label.

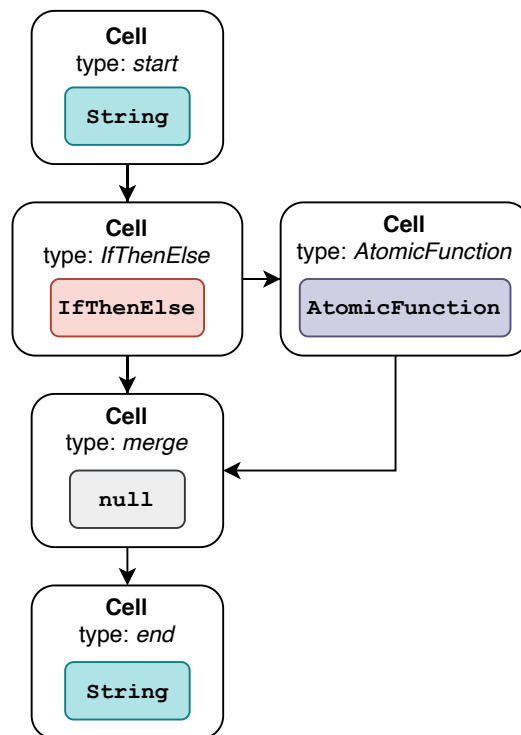


Figure 3.4: A graph showing cells and their associated user objects

The following table shows the implemented `Cell` types which are used as visual elements representing a node in the graph. As well as a short description, the type of the associated user object is indicated.









Cell	Description	User Object
 Start	Defines the workflow's entry point	String (for the label)
 End	Defines the workflow's termination	String (for the label)
 Function	An atomic function	AtomicFunction *
 If-Then-Else	A mutual exclusive condition	IfThenElse *
 Switch	A multi condition	Switch *
 Merge	An element for merging previous branches	null (none)
 Parallel	A container, where each child cell is meant to be executed in parallel.	Parallel *
 ParallelFor	A container, where each child cell is meant to be executed in a parallel for loop.	ParallelFor *

Table 3.1: Implemented Cells, *: AFCL Object

Graph Validation

A DAG which represents a *valid* AFCL workflow always fulfills the following constraints:

- There exists exactly one **Start**
- There exists exactly one **End**
- There exist no cycles
- Every **Cell** has a path to **Start**
- Every **Cell** has a path to **End**
- **Function**, **If-Then-Else**, **Switch**, **Parallel**, **ParallelFor** and **End** have exactly one incoming edge
- **Start**, **Function**, **Parallel**, **ParallelFor** and **Merge** have exactly one outgoing edge
- **If-Then-Else** has exactly two outgoing edges (then, else)
- **Switch** has multiple outgoing edges
- **Merge** has multiple incoming edges

In addition to the DAG-constraints above, the following additional constraints are needed:

Compound constructs are visualized as container which enclose their child nodes. It has to be ensured that there exists no direct connection from a cell inside a container to a cell outside that container.

- Both **Cells** taking part in a direct connection, must have the same nesting level, what means they must have the same parent cell

In an AFCL workflow, a function's name has to be unique:

- A **Cell**'s label has to be unique in the workflow

These constraints are enforced at two different stages: During editing, more specifically when adding and connecting vertices, and before saving or adapting a workflow. Due the fact that the user cannot even create an invalid workflow in terms of execution flow, efficiency increases and errors in the corresponding backend services will be minimized.

Graph Encoding

The visual representation of the workflow and the included data targets human-readability. This data needs to be put into another, proper format for storing. `mxGraph` provides built-in support for converting the visual representation to XML. An advantage of using XML over JSON is that constructor names (class names) are mapped to the XML node names, no additional property is required. What makes XML also very convenient is Java's excellent native support for XML and XPath on the backend side. The XML-encoded workflow is converted on the backend-side internally to an AFCL workflow.

```
<Workflow name="Untitled">
  <Array as="dataIns" />
  <Array as="dataOuts" />
  <Array as="body">
    ...
    <Cell id="4" style="fn" type="AtomicFunction" ...>
      <AtomicFunction name="f1" type="f1Type" as="value">
        <Array as="properties" />
        <Array as="constraints" />
        <Array as="dataIns" />
        <Array as="dataOuts" />
      </AtomicFunction>
      <mxGeometry x="40" y="40" width="88" height="32" as="geometry" />
    </Cell>
    ...
  </Array>
</Workflow>
```

Listing 3.1: Example of a XML-encoded Workflow.

Graph Layouting

When a user loads an AFCL file into the editor, the cells have to be arranged in a way such that the visualization makes sense. There exists no information in AFCL workflows about any visual properties.

Fortunately, `mxGraph` ships with multiple layouting algorithms. For AFCL graphs, the `mxHierarchicalLayout` algorithm fits best to arrange the elements. This worked well, unless the graph has some container nodes, which the layouting algorithm is unable to handle. To overcome this problem, a custom layouting algorithm was developed, which layouts all container elements using `mxHierarchicalLayout` recursively, beginning at the innermost to the outermost container.

Graph Alternatives

One of the more difficult tasks was to choose a proper library for graph drawing. Since the graph drawing and visualization is one of the most important features of the frontend, a careful choice has to be made. Multiple excellent JavaScript graph drawing libraries currently exist, but not all of them are suitable for the requirements of this project.

Below is a list of tested and researched JavaScript graph libraries with additional information on why they were insufficient for the project.

- **jsPlumb** is a library of great quality and looks like it would fulfill all requirements for graph drawing and user interaction out of the box. It has very good documentation, and a lot of examples. Furthermore its animations and drag and drop handling are outstandingly good. Unfortunately this software is closed-source and a license costs a few thousand dollars.
- The same goes for **Rappid** (formerly known as JointJS) and **GoJS**, which is even more expensive.
- **diagram-js** is one of the newer diagram libraries. It seems to be a good candidate to draw BPMN graphs.¹⁰ The lack of appropriate documentation - even after four years of existence (there exists an open issue on github¹¹ for this) - is still an issue.
- **Cytoscape.js** is older than `mxGraph` but the development and the community is not less active. This library has very good documentation and a lot of demos are provided. But it offers fewer examples and "out-of-the-box" features for modeling flowcharts than `mxGraph` does.

¹⁰<http://www.bpmn.org>

¹¹<https://github.com/bpmn-io/diagram-js/issues/78>

- **vis.js** is a star on npmtrends.¹² It has very smooth animations when manipulating the graph. There exists only one example of graph manipulation on its documentation, as it is mainly used for visualization only.
- **Sigma js** is a tiny library with a small documentation. It focuses on displaying and visualizing graphs, not on modeling them. This part would have to be developed manually. Furthermore, the last activity on this project on github was two years ago.

3.2.2 Backend

The backend of the application is built entirely with Java, using Maven as package manager and Apache Tomcat as Servlet Implementation.

API

The API to interact with the frontend or other clients is based on HTTP and it is inspired by the principles of REST and RPC.

RPC-based APIs are great for actions (that is, procedures or commands). REST-based APIs are great for modeling your domain (that is, resources or entities), making CRUD (create, read, update, delete) available for all of your data. [10]

Function data is stored server-side, thus its API is a REST-based implementation. Workflows are stored in files on a user's disk and workflow adaptation is a "remote command" which sends data to the server and asks it to process and return the result. Therefore the API to adapt workflows is RPC-based. All API requests use JSON as format for receiving and returning data.

Decoding Frontend Data

Since the frontend encodes the visual representation of the workflow (the graph) in XML, this data has to be converted to AFCL Java objects in order to process it with the AFCL Java API. The excellent out-of-the support of Java for XML documents turned out to

¹²<https://www.npmtrends.com/vis>

be very convenient. With the power of XPath expressions, it was quite easy to operate on the XML structure and extract the relevant information to convert the XML-based workflow into an AFCL workflow. Listing 3.2 shows an example of a simple XPath expression, which finds all nodes whose node name equals `AtomicFunction`, having a `name` attribute whose value equals `'getFlight'`.

```
//AtomicFunction[@name='getFlight']
```

Listing 3.2: Example of an XPath expression.

Persistence

The storing of function data is abstracted from the user and persisting of the data is handled internally. For this purpose a *repository interface* was created on the backend side to be capable of supporting any data source.¹³

The repository which implements the interface in this thesis, stores the serialized data into a file. This could be easily replaced with any other implementation, for example an implementation which stores the data in a DBMS.

```
public interface Repository<T> {  
    public Collection<T> findAll();  
    public T findOne(String id);  
    public void add(T obj);  
    public void remove(T obj);  
    public void remove(String id);  
    public void update(T obj);  
}
```

Listing 3.3: Repository Interface

¹³<https://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html>

Adaptation

Modifications on a workflow, programmatical or manual, consist always of operations modifying control structure (execution flow) and operations modifying data input and/or output elements associated to control structure elements (data flow). Most of the time, a modification on a workflow's execution flow (e.g. by adding, modifying or removing control structure) entails modifications on dependent data elements.

In order to support more than one specific kind of workflow adaptation, a service providing general workflow operations has been developed. The service is designed to be as generic as possible, such that certain functions can be reused later or in other projects for any other kind of workflow modifications.

Part of these functions can be found in A.

I think the following paragraph including Fig. 3.5 should go in system architecture for the backend (What did you do?). Here you write details how did you solve that with the three procedures (How did you do?)

The concrete workflow adaptation developed in this thesis is to divide `ParallelFor` elements into multiple sections in a `Parallel` element. Note that the `loopCounter` property on the newly created `ParallelFor` elements is adjusted such that the iterations of the original loop are distributed onto the newly created ones.

Figure 3.5 visualizes the core part of this adaptation in its simplest form. This adaptation is useful when the number of iterations reaches the maximum number of concurrent function invocations of the FaaS provider. To be able to scale independently of this limitation, the execution of the inner loops of the adapted workflow could be distributed over multiple FaaS systems. In addition, this adaptation can save costs and may increase performance.

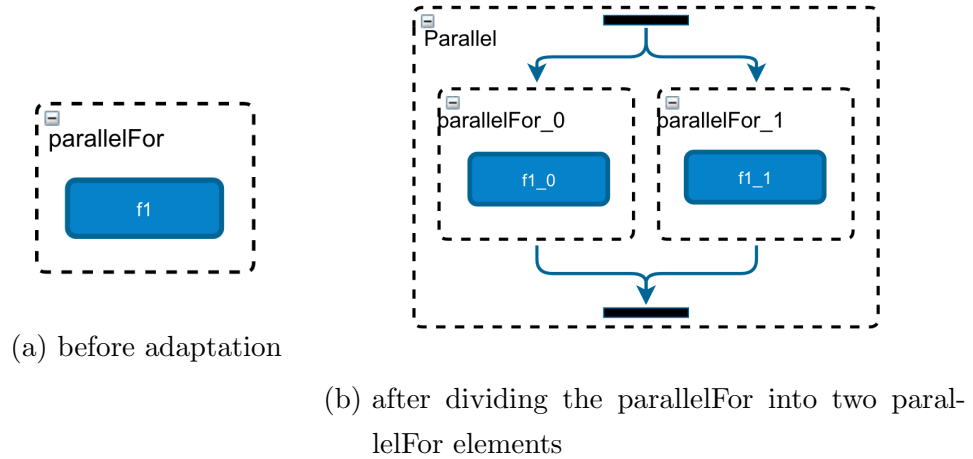


Figure 3.5: workflow adaptation

While the mentioned steps modify the workflow's execution flow, it is important that the data-flow has to be updated according to the new structure, to ensure the workflow's execution works as it did before the modification. This includes the creation and rearrangement of associated **DataIns** and **DataOuts** elements, as well as updating the **source** property on all affected elements.

The whole procedure can be structured into three logical phases, consisting of the following tasks.

Phase A - Preparation

1. get **ParallelFor**, which will be divided
2. create new **Parallel**
3. replace **ParallelFor** with **Parallel**
4. move data items (**dataIns**, **dataOuts**) from **ParallelFor** to created **Parallel**
5. generate data flow from new **Parallel** to **ParallelFor**
6. create a new **DataOuts** to collect output data from inner branches

Phase B - Dividing

7. for each branch:

- a) clone `ParallelFor`
- b) set `LoopCounter` of cloned element according to input
- c) rename all functions in created branch (includes adjusting **source** of all subsequent affected `DataIns` and `DataOuts`),
- d) add output data to the prepared `DataOuts` from 6.

Phase C - Adjusting

- 8. set prepared `DataOuts` to created `Parallel`
- 9. adjust source of all subsequent `DataIns/DataOuts`

it will be nice to give an example of renaming the inner functions, adding prefixes/-suffixes

4 Web Interface

Web interface is missing on the architecture figure.

In this chapter the user interface and all features of the web application are described in detail.

The layout of the web interface is split into a navigation on the left and the main view. The user can change the main view by selecting an item in the left navigation.

In the following sections, the three main components are described.

I guess the three components are functions, editor and settings, which are accessible through Dashboard. So, better not to put 4.1 dashboard.

4.1 Dashboard

The dashboard is the entry point where the user lands after accessing the app. The purpose of this component is to give the user a quick overview of the application, provide short informational texts and links to the specific components.

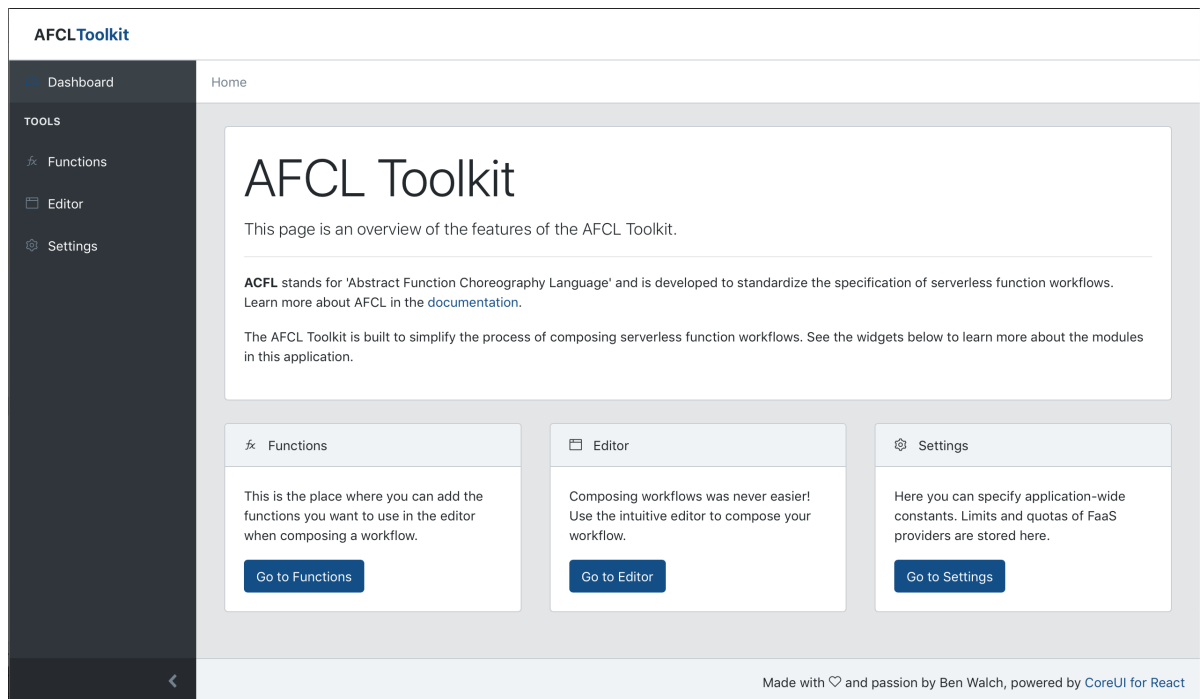


Figure 4.1: the dashboard

4.2 Functions

You need to connect functions in backend architecture in Section 3.1. That you use a repository so that the user can create a workflow from existing functions? How do you load a workflow if there are no existing functions?

The function repository is needed, to know about available functions when composing a workflow. A list of all functions stored is presented to the user. An item in this list holds the following data: *name*, *type* and *provider*.

The user has the option to remove certain items in the list by clicking on the button with the "trash" icon. Before a function is really deleted, the user has to confirm the deletion. New functions can be added by clicking the button "add function" on the upper left.

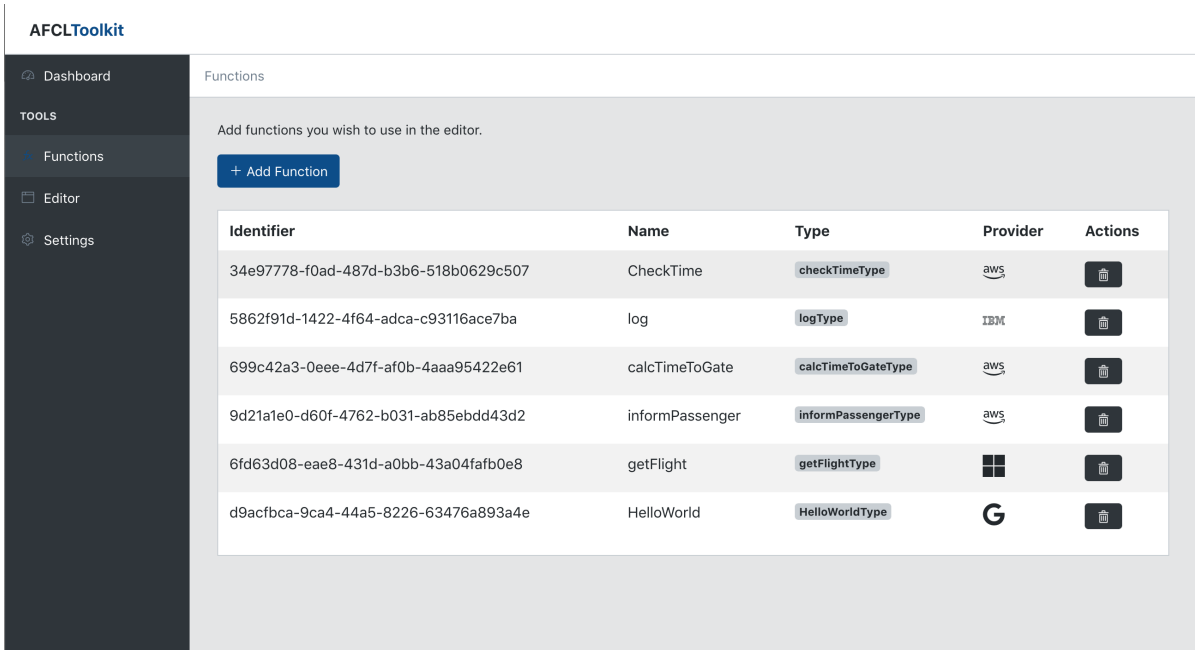


Figure 4.2: the function repository

4.3 Editor

The interface responsible for composing workflows is divided into two parts. On the left side is the editor, on the right side is the property view. The editor consists of a sidebar (second bar left of figure 4.3), the drawing area (center of figure 4.3) and a toolbar (top of figure 4.3).

In the following, the terms *node*, *cell* or *vertex* refer all to the same meaning: the visual element which represents a node in the graph.

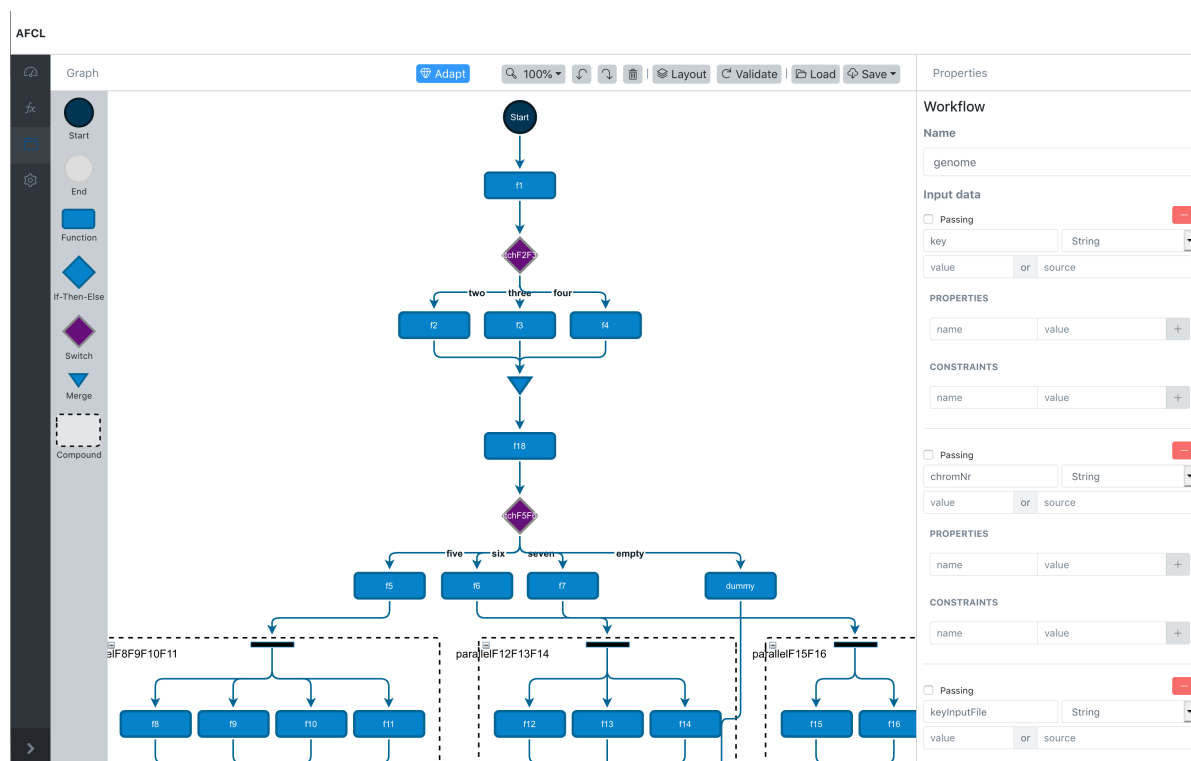


Figure 4.3: the editor

4.3.1 Drawing Area

The drawing area is the part where probably the most user interaction occurs.

Cells can be arranged and connected to each other by drag and drop. The possible connection points (ports) of a cell will be displayed when hovering over it. These ports are input and output for most of the functions. When hovering a port, the port name appears in the tooltip. To create a connection between two cells, the output port of the cell can be dragged to an input port of another cell.

During the connecting process, the application takes care that no invalid graph can be created by validating all constraints mentioned in section 3.2.1. on-the-fly. If the result of the validation is invalid, the preview for the connection being edited turns red, and the target port of the connection being edited will be disabled, removing the possibility to drag and drop on that port.

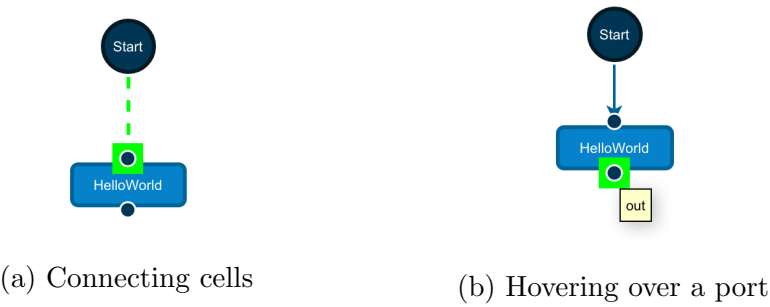


Figure 4.4: graph modeling

4.3.2 Sidebar

All available graphical elements for composing a workflow are shown in the sidebar. The elements have already been technically described in table 3.1.

The user can choose an element from the sidebar by clicking on it, which results in placing a cell of the selected type in the drawing area. For functions or compound constructs, a drop-down list opens with the available choices.

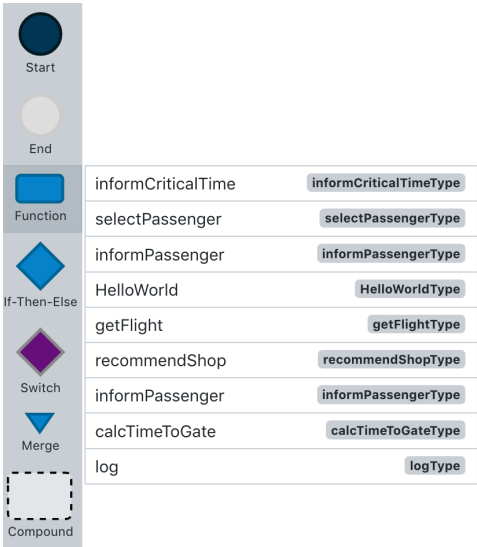


Figure 4.5: the sidebar with a dropdown open

4.3.3 Property View

When the user selects a cell in the drawing area, the property view updates. The property view has two sections. The first (upper) section displays properties of the AFCL object associated with the selected cell. The second (lower) section displays properties of the selected cell itself. Depending on the cell's type (function, control-structure, edge, ...) the displayed information differs.

In the first section, the user can here edit all properties (data input, a.k.a *dataIns*, data output, ...) of the AFCL object associated with the selected cell. Changes of fields inside the property view execute immediately, there is no need to confirm or save those. The corresponding properties for each AFCL object are not described here, detailed information about them is available at [1].

The cell properties in the second section worth mentioning are *name* and *type*. The remaining properties are mainly for informational and debugging purposes and won't be explained in detail.

4.3.4 Toolbar

To make all actions available to the user, a toolbar is displayed in the top area of the editor. Each button of the toolbar represents an action, which are described in detail in detail below.

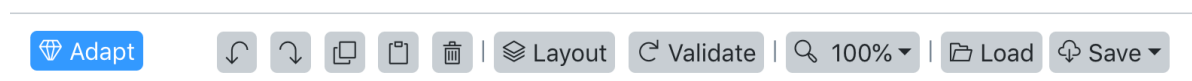


Figure 4.6: the toolbar

Zoom

The zoom button offers the opportunity to change the scale of the current view in the drawing area. The user can select out of the following values: 50%, 75%, 100%, 125%, 150%. The button's label is always the current selected value.

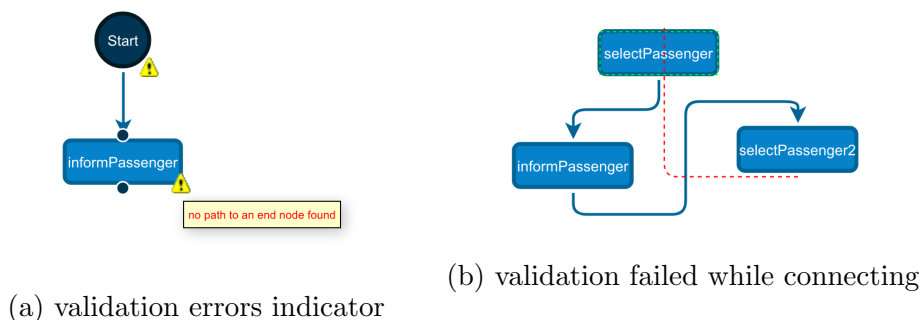




Figure 4.7: graph validation


Action History (Undo & Redo)

With the buttons ↶ (undo) and ↷ (redo), the user can undo or redo actions he/she made in the drawing area. These actions are also available as keyboard shortcuts: **Ctrl-Z** or **Ctrl-Y**, respectively.

Clipboard (Copy & Paste)

The editor includes a clipboard. It is possible to transfer selected cells using the  button (copy) or the keyboard shortcut **Ctrl-C** to the clipboard. The contents of the clipboard can then be pasted by using the  button (paste) or the keyboard shortcut **Ctrl-V**.

Delete

Selected cells can be removed by clicking the  button (delete). Alternatively, this can be done with the keyboard shortcut **Backspace**.

Validation

The action behind the "Validate" button validates the constraints described in 3.2.1 for each cell in the drawing area. Invalid cells will be marked with a yellow icon. An additional error message is shown in the tooltip when hovering over that icon.

Load & Save

Workflows created by AFCL ToolKit (XML) or even existing AFCL workflows (YAML/JSON) can be loaded in order to visualize or do further editing. To load a workflow, the user can click on the "Load" button in the toolbar and choose the desired file in the dialog.

To save a workflow, the user can click on the "Save" button in the toolbar. This results in a download offered by the tool. The user can then choose the target location. When saving a workflow, the desired format can be selected out of XML, JSON or YAML.

With XML, additional graphical information is stored with the serialized workflow, what means that the workflow will be exactly in the same state when opened later.

JSON and YAML deliver an AFCL compliant workflow.

4.4 Settings

too short for a subsection. Give some examples of such constants, maybe also with a figure.

The purpose of the Settings component is to provide an interface to view and adjust application-wide constants.

5 Evaluation

This section evaluates AFCL ToolKit and compares the effort for creating and adapting AFCL workflows using the tool with the effort needed to write or adapt the AFCL workflow manually in a text file or using the Java API.

The *Gate Change Alert* (GCA), *Anomaly Detection* (AD) and *Genome* (GEN) AFCL workflow examples from [1] are used as evaluation models.

It has to be noted, that only the time it took to get the final result - which is an AFCL workflow in a YAML file - is evaluated. The learning time to understand the AFCL function choreography system, the Java API or to become familiar with AFCL Toolkit is not considered in this study.

5.1 Methodology

In this section, the methodology for the evaluations in sections 5.2 and 5.3 is explained:

One computer science student learned the workflow language AFCL, its Java-based API and how to use AFCL ToolKit. Then the different workflows were composed (section 5.2) or adapted (section 5.3) by

- (1) editing the text file manually (writing YAML by hand),
- (2) using the AFCL Java API,
- (3) using AFCL ToolKit.

The results were then compared to each other.

5.2 Composing Workflows

In this section, the development effort of the three example workflows is measured, using the three introduced methods.

put names for methods (manual, AFCL API, and AFC-Toolkit).

Write some % that your toolkit generates workflows the fastest. Explain why manual is much slower (e.g. in lines of code)

It took approximately 55 minutes to compose the GCA workflow with method (1), approximately 25 minutes with method (2) and approximately 22 minutes with method (3).

For the AD workflow it took approximately 75 minutes with method (1), approximately 42 minutes with method (2) and approximately 37 minutes with method (3).

The GEN workflow took approximately 49 minutes with method (1), approximately 24 minutes with method (2) and approximately 20 minutes with method (3).

Figure 5.1 shows a visualization of the evaluation.

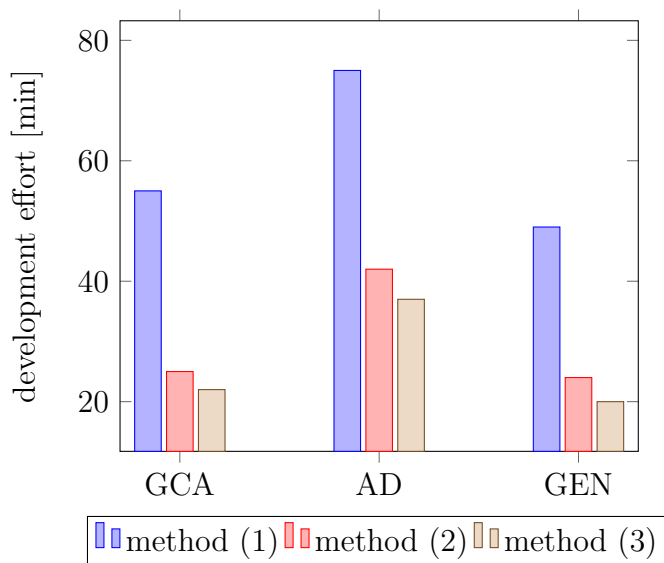


Figure 5.1: Evaluated development effort for creating GCA, AD and GEN AFCL workflows using different methods.

method (1): edit text file manually, method (2): AFCL Java API, method (3): AFCL ToolKit

This evaluation shows, that using AFCL ToolKit can speed up the process for creating AFCL workflows significantly.

A big benefit of using AFCL ToolKit over using the Java API is, that anyone with knowledge about AFCL could create the workflow, there is no need to be a programmer. Additionally, human errors are minimized. According to [7] the error rate of a developer is between 1.5% and 5% in average. With AFCL ToolKits' immediate validation while creating a workflow, such errors are minimized even further.

5.3 Adapting Workflows

Start here - the full benefit / contribution of the AFCL Toolkit is visible in FC adaptation.

In this section, the effort required for doing a workflow adaptation as described in 3.2.2 is measured.

For this evaluation, it was sufficient to only measure the effort needed to divide a loop manually into two loops. Since the procedure would be the same for every additional loop, the total approximated effort can then be determined by multiplying the effort which was needed for a divide into two loops, with the number of divides. In the following, we will only consider the GCA workflow example, since this includes one `ParallelFor` loop. The effort is also dependent of the number of affected data elements in the subsequent functions, which have to be adjusted. In the GCA workflow, there are twelve data elements which need to be updated after a divide.

Can you estimate some equation to estimate the speedup? In case of more complex nesting in the loop, AFCL toolkit has few changes only, while other methods need to copy and update many.

To divide the loop of the GCA workflow into two loops, it took approximately 27 minutes using method (1), approximately 19 minutes using method (2) and 1 minute using method (3).

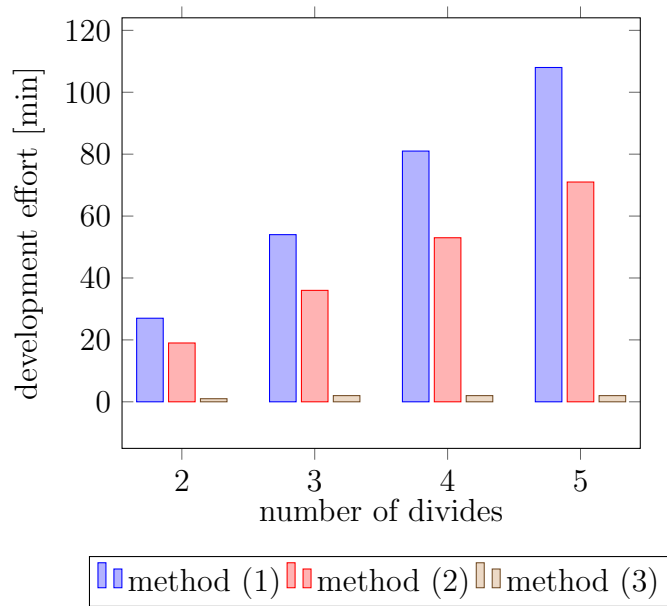


Figure 5.2: Evaluated development effort for adapting workflows using different methods.
method (1): edit text file manually, method (2): AFCL Java API, method (3): AFCL ToolKit

6 Conclusion

With AFCL ToolKit, a system was developed that greatly simplifies the creation and adaptation of AFCL workflows. The implemented system fulfills the requirement of modeling workflows at a high level of abstraction by providing an editor where workflows can be created and edited in a WYSIWYG manner. It even goes further by providing additional user actions like clipboard, versioning, auto-layout and validation, which make the creation process more convenient. The system also can load existing AFCL workflows for visualization and further editing and supports three different formats for exporting workflows.

We have shown that the system with its benefits leads to an enormous speed up for workflow development and adaption. With these possibilities, AFCL is no longer reserved exclusively for developers. By using this tool, virtually anyone is able to develop a workflow.

do not use separate section for future work. Either at least two or zero sections in the chapter.

6.1 Future Work

- additional information in tooltips
- auto-complete for `source` fields
- visualize data flow

A AFCL API functions

In the following paragraphs of this section, terms highlighted with a mono-spaced font refer to classes or class properties of the AFCL Java API.

The AFCL Java API in its current state provides classes which define function and control-structure objects, as well as data-flow objects of an AFCL workflow. They can be arranged and nested in a tree structure in order to model the execution flow.

However, the API has its limitations. One is that a `Function` does not have a reference to its *parent object*. The *parent object* can be either a `Workflow` or `Compound`, which has its own implementation to access its enclosed functions - there is no common super class or interface providing a common method. The same problem occurs on `DataIns`, `DataOuts` and `DataOutsAtomic`, they do have properties (e.g. `source`) in common, but each of them has its own implementation to access them. This is also the case for `AtomicFunction` and `Compound` and the `dataIns` property.

Furthermore, tasks like iterating over all `Function` objects in a `Workflow` are not supported.

These limitations make it challenging to perform automated modifications while keep the code generic. During development, it turned out that the following tasks were required frequently while modifying a workflow programmatically. We define these task as *general adaptation tasks*:

- get a `Function` by its name
- get the *parent object* of a `Function` (`Workflow` or `Compound`)
- get the `List` which contains a given `Function`
- get all `DataIns` and `DataOuts` which use a given `Function` as source

- traverse a `Workflow`, in particular, iterate over all `Function` objects inside a `Workflow`

The generic requirement was achieved on the one hand by using *Reflection*, on the other hand by developing a generic *traverse function*, shown in listings A.1 and A.2.

```
public static void traverseWorkflow(Workflow wf, BiConsumer<Function, Object> consumer) {
    traverseFunctions(wf.getWorkflowBody(), consumer, wf);
}
```

Listing A.1: Traverse workflow

```
public static void traverseFunctions(List<Function> functionsList, BiConsumer<Function, Object> consumer) {
    if (functionsList == null) {
        return;
    }
    for (Function fn : functionsList) {
        consumer.accept(fn, currentParent);
        if (fn instanceof IfThenElse) {
            traverseFunctions(((IfThenElse) fn).getThen(), consumer, fn);
            traverseFunctions(((IfThenElse) fn).getElse(), consumer, fn);
        }
        if (fn instanceof Switch) {
            for (Case c : ((Switch) fn).getCases()) {
                traverseFunctions(c.getFunctions(), consumer, fn);
            }
        }
        if (fn instanceof Parallel) {
            for (Section s : ((Parallel) fn).getParallelBody()) {
                traverseFunctions(s.getSection(), consumer, fn);
            }
        }
        if (fn instanceof ParallelFor) {
            traverseFunctions(((ParallelFor) fn).getLoopBody(), consumer, fn);
        }
    }
}
```

```
}
```

Listing A.2: Traverse functions

As the reader may have noticed, the traverse function accepts Java's `BiConsumer` as second argument, which makes it very versatile in its usage. The given consumer operation - which can be a function reference or a lambda expression - is executed for every function element in the workflow, providing the element itself and its *parent object* as arguments on traversal. This is indeed very powerful, reduces LOC, therefore improves readability, maintainability and efficiency. The only drawback is that all functions are always visited, because there is no proper way to break out of a lambda expression. There exist approaches to break out by throwing an `Exception`, however this is considered to be bad practice.

For example, getting a function by its name, can be achieved as showed in listing A.3. Note that mutating variables in lamda expressions is not thread-safe, so an `AtomicReference` is used.

```
final AtomicReference<Function> fRef = new AtomicReference<>();
traverseFunctions(fnList, (fn, parentObj) -> {
    if (fn.getName() != null && fn.getName().equals(name)) {
        fRef.set(fn);
    }
});
// do something with found function in fRef.get();
```

Listing A.3: get a function by its name

Bibliography

- [1] *AFCL - Abstract Function Choreography Language*. URL: <http://dps.uibk.ac.at/projects/afcl/> (visited on 04/10/2020).
- [2] Paul Castro et al. “The Rise of Serverless Computing”. In: *Commun. ACM* 62.12 (Nov. 2019), pp. 44–54. ISSN: 0001-0782. DOI: 10.1145/3368454.
- [3] Erwin van Eyk. “The design, productization, and evaluation of a serverless workflow-management system”. 2019.
- [4] Cloudflare Inc. *What Is Serverless Computing?* URL: <https://www.cloudflare.com/learning/serverless/what-is-serverless/> (visited on 03/08/2020).
- [5] Ali Kanso and Alaa Youssef. “Serverless: Beyond the Cloud”. In: *Proceedings of the 2nd International Workshop on Serverless Computing*. WoSC ’17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 6–10. ISBN: 9781450354349. DOI: 10.1145/3154847.3154854.
- [6] JGraph Ltd. *mxGraph User Manual*. URL: <https://jgraph.github.io/mxgraph/docs/manual.html> (visited on 04/07/2020).
- [7] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. 2004. ISBN: 978-0735619678.
- [8] Nick Russell et al. *Workflow Control-Flow Patterns: A Revised View*. 2006. URL: <http://workflowpatterns.com/documentation/documents/BPM-06-22.pdf>.
- [9] Neil Savage. “Going Serverless”. In: *Commun. ACM* 61.2 (Jan. 2018), pp. 15–16. ISSN: 0001-0782. DOI: 10.1145/3171583.
- [10] Phil Sturgeon. *Understanding RPC Vs REST For HTTP APIs*. URL: <https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/> (visited on 04/29/2020).