

DISCOPAR: A Visual Reactive Flow-Based Domain-Specific Language for Constructing Participatory Sensing Platforms

Jesse Zaman

The evolution of the smartphone as a general computing platform combined with the rich sensing functionalities that it has acquired in recent years, have led to a new collective data gathering paradigm called participatory sensing. Participatory sensing is the driving technology behind so-called citizen observatories; i.e. a set of cloud-based software tools that are used to gather, analyse and visualise data by a group of citizens that share some collective concern. Participatory sensing is often used in so-called campaigns. A campaign is a collective data gathering effort that is delimited in space and/or time.

Today citizen observatories have to be developed from scratch for each application domain, meaning that deploying a new citizen observatory is nothing less than a complex cloud-driven software engineering project that is extremely labour-intensive precisely because of its technical complexity. Despite an overwhelming demand for such platforms, they are thus beyond the reach of most societal stakeholder groups.

What is needed is a generic approach towards reusable and reconfigurable citizen observatories, i.e. a citizen observatory meta-platform that can be used by stakeholders to create new and adapt existing citizen observatories. Thus, apart from the technical design challenges, a key requirement of such a meta-platform is that it is easily accessible by societal stakeholders and communities. Deploying a new citizen observatory and setting up campaigns through the meta-platform should therefore be possible without or with only very limited programming skills.

In this dissertation, we present DISCOPAR (Distributed Components for Participatory Campaigning), a new visual reactive flow-based domain-specific programming language created specifically to hide the non-essential complexity of citizen observatories from the end-user, and to present only concepts that are truly relevant to their domain. DISCOPAR is used throughout the meta-platform to enable end-users to construct every part of a citizen observatory: the mobile data gathering app, server-side data processing, and web-based visualisations can all be set up using a single visual language, thereby greatly increasing the accessibility by end-users.

We validate our citizen observatory meta-platform and the DISCOPAR language - in terms of expressiveness, suitability and usability - through experiments both in laboratory as well as in real-world conditions. We demonstrate expressiveness by creating three radically different citizen observatories and test the suitability and usability during real-world experiments performed by different groups of people without any programming knowledge.



DISCOPAR
A Visual Reactive Flow-Based Domain-Specific Language
for Constructing Participatory Sensing Platforms

DISCOPAR

A Visual Reactive Flow-Based Domain-Specific Language for Constructing Participatory Sensing Platforms

Dissertation Submitted for the Degree of Doctor of Philosophy in Sciences

Jesse Zaman

September, 2018

Promotor:

Prof. Dr. Wolfgang De Meuter



SOFTWARE LANGUAGES LAB
DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF SCIENCE AND BIO-ENGINEERING SCIENCES



DISCO PAR



**A Visual Reactive Flow-Based Domain-Specific
Language for Constructing Participatory
Sensing Platforms**

Jesse Zaman

September 6, 2018

© 2018 Jesse Zaman

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 978 94 9231 289 1
NUR 989

All rights reserved. No part of this publication may be produced in any form by print, photoprint, microfilm, electronic or any other means without permission from the author.

ABSTRACT

The evolution of the smartphone as a general computing platform combined with the rich sensing functionalities that it has acquired in recent years, have led to a new collective data gathering paradigm called participatory sensing. Participatory sensing is the driving technology behind so-called *citizen observatories*; i.e. a set of cloud-based software tools that are used to gather, analyse and visualise data by a group of citizens that share some collective concern. Participatory sensing is often used in so-called *campaigns*. A campaign is a collective data gathering effort that is delimited in space and/or time.

Today citizen observatories have to be developed from scratch for each application domain, meaning that deploying a new citizen observatory is nothing less than a complex cloud-driven software engineering project that is extremely labour-intensive precisely because of its technical complexity. Despite an overwhelming demand for such platforms, they are thus beyond the reach of most societal stakeholder groups.

What is needed is a generic approach towards reusable and reconfigurable citizen observatories, i.e. a citizen observatory *meta-platform* that can be used by stakeholders to create new and adapt existing citizen observatories. Thus, apart from the technical design challenges, a key requirement of such a meta-platform is that it is easily accessible by societal stakeholders and communities. Deploying a new citizen observatory and setting up campaigns through the meta-platform should therefore be possible without or with only very limited programming skills.

In this dissertation, we present DISCOPAR (Distributed Components for Participatory Campaigning), a new visual reactive flow-based domain-specific programming language created specifically to hide the non-essential complexity of citizen observatories from the end-user, and to present only concepts that are truly relevant to their domain. DISCOPAR is used throughout the meta-platform to enable end-users to construct every part of a citizen observatory: the mobile data gathering app, server-side data processing, and web-based visualisations can all be set up using a single visual language, thereby greatly increasing the accessibility by end-users.

We validate our citizen observatory meta-platform and the DISCOPAR language – in terms of expressiveness, suitability and usability – through experiments both in laboratory as well as in real-world conditions. We demonstrate expressiveness by creating three radically different citizen observatories and test the suitability and usability during real-world experiments performed by different groups of people without any programming knowledge.

SAMENVATTING

De evolutie van de smartphone als computerplatform, gecombineerd met de rijke technologische functies die het de afgelopen jaren heeft verworven, heeft geleid tot een nieuw paradigma voor gegevensverzameling dat *participatief meten* wordt genoemd. Participatief meten wordt vaak gebruikt in *campagnes*, het verzamelen van data door een groep mensen in een specifiek gebied en/of tijdsinterval.

Participatief meten is de motor achter zogenaamde *burgerobservatoria*; een reeks ICT-hulpmiddelen om gegevens te verzamelen, analyseren en visualiseren met als doel de levenskwaliteit van burgers te verbeteren. Tegenwoordig moeten burgerobservatoria voor elk toepassingsgebied vanaf nul worden ontwikkeld, waardoor een nieuw burgerobservatorium creëren uiterst moeilijk en arbeidsintensief blijft. Ondanks een overweldigende vraag naar dergelijke platforms, zijn ze dus buiten het bereik van de meeste maatschappelijke belanghebbenden.

Bijgevolg is een generieke benadering van herbruikbare en (her)configureerbare burgerobservatoria meer dan nodig. Dit proefschrift introduceert een burgerobservatoria *metaplatform*, een platform dat belanghebbenden kunnen gebruiken om burgerobservatoria te creëren. Een van de grootste uitdagingen van dit metaplatform is de toegankelijkheid verzekeren voor betrokken personen en gemeenschappen. Het creëren van een burgerobservatorium en het opzetten van campagnes moet daarom mogelijk zijn zonder of met beperkte programmeervaardigheden.

Om dit mogelijk te maken, hebben we DISCOPAR (Distributed Components for Participatory Campaigning) gecreëerd. DISCOPAR is een nieuwe visuele, reactieve, flow-gebaseerde, domeinspecifieke programmeertaal die specifiek de ongewenste complexiteit van burgerobservatoria verbergt voor de eindgebruiker en alleen concepten presenteert die relevant zijn voor hun domein. DISCOPAR wordt overal in het metaplatform gebruikt om eindgebruikers in staat te stellen elk deel van een burgerobservatorium te bouwen. De mobiele app voor het verzamelen van gegevens, de gegevensverwerking op de server, en ook de webgebaseerde visualisaties kunnen allemaal worden opgezet met behulp van één visuele taal. Hierdoor vergroot de toegankelijkheid voor eindgebruikers aanzienlijk.

Om de expressiviteit en correctheid van het herconfigureerbare platform aan te tonen, hebben we drie radicaal verschillende burgerobservatoria gecreëerd en deze getest in zowel het laboratorium als in reële omstandigheden. De toegankelijkheid van het metaplatform werd getest tijdens experimenten uitgevoerd door verschillende groepen bestaande uit mensen zonder programmeerkennis.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my promotor, Prof. Dr. Wolfgang de Meuter. Many thanks for your advice and encouragement throughout all these years. My atypical research topic sometimes made me feel like a lone wolf at SOFT. Luckily, there was this other Wolf whose occasional praising made me feel part of the pack.

I would also like to thank the members of my jury: Prof. Dr. Ann Nowé, Prof. Dr. Coen De Roover, Prof. Dr. Cathy Macharis, Prof. Dr. em. Theo D’Hondt, Prof. Dr. Boris Magnusson, and Dr. ir. Thomas Springer. Thank you for critically reading my dissertation and helping to improve the quality of this text.

I would like to express my very great appreciation to Dr. Jens Nicolay for his valuable and constructive suggestions that helped to improve the structure of this dissertation.

I also wish to acknowledge the excellent work environment provided by all (former) members of the Software Languages Lab. My special thanks go to my colleagues with whom I started this journey together: Simon, Janwillem, and Nathalie.

I am particularly grateful for the support of my friends and family, with an additional shout-out to the “T-Time”-crew: Dries, Jonas and Caro, Dieter and Emma, and Karine and Gie. An army marches on its stomach, so special thanks to my mother for keeping me well-provisioned during the entire Battle of the PhD.

I also want to express my gratitude to Jessica for being a great friend. Our weekly get-togethers are always something I look forward to.

Last but not least, I would like to thank Ayla for showing me what love is all about. In the grand scheme of things, our amazing adventure is only just beginning. I am looking forward to an equally amazing future, without doubt accompanied by our four-legged friends.

This research was funded by a PRFB grant from Innoviris, the Brussels Institute for Research and Innovation.

TABLE OF CONTENTS

1	Introduction	1
1.1	Problem Statement	3
1.2	Research Vision	4
1.3	Methodology	5
1.4	Supporting Publications and Technical Contributions	6
1.5	Dissertation Outline	8
2	Participatory Campaigning and Citizen Observatories	11
2.1	Sensors Galore	12
2.2	Participatory Sensing	14
2.2.1	Ad-Hoc Participatory Sensing Systems	15
2.2.2	Reusable Participatory Sensing Systems	17
2.2.3	PS Research Challenges	19
2.3	Participatory Campaigning	20
2.3.1	Campaign Protocol	21
2.3.2	Campaign Definition	23
2.3.3	Campaign Lifecycle	23
2.4	Citizen Observatories	25
2.4.1	Citizen Observatory Stakeholders	27
2.4.2	Research in Citizen Observatories	28
2.4.3	Research Questions	29
2.5	Vision: Citizen Observatory Meta-Platform	31
2.6	Conclusion	33
3	Towards a Citizen Observatory Meta-Platform: Requirements	35
3.1	Citizen Observatory Architecture	36
3.2	Meta-Platform Stakeholders	37
3.3	Meta-Platform Requirements	38

3.3.1	User Level Requirements	38
3.3.2	Implementation Level Requirements	40
3.4	The Right Tool for the Job	42
3.5	Flow-Based Programming	43
3.5.1	Dataflow Programming	44
3.5.2	Flow-Based Programming Characteristics and Classification	45
3.6	Reactive Programming	47
3.7	Visual Programming Languages	48
3.7.1	Visual Data Flow Programming Languages	49
3.8	Domain-Specific Languages	50
3.8.1	Domain-Specific Modelling	50
3.9	Related Work	51
3.10	Conclusion	55
4	Language Concepts of DISCOPAR	57
4.1	Programming with DISCOPAR	58
4.2	Concepts and Terminology	60
4.2.1	Components	61
4.2.2	Processes	61
4.2.3	Connections	62
4.2.4	Ports	63
4.2.5	Information Packets	64
4.2.6	Graphs	65
4.3	DISCOPAR ^{DE}	66
4.3.1	Component Menu	67
4.3.2	Canvas	68
4.3.3	Process Configuration	70
4.3.4	Graph Validation Indicator	71
4.3.5	DISCOPAR ^{DE} as Live Programming Environment	72
4.4	Facilitating End-User Programming	73
4.5	Conclusion	75
5	Constructing Citizen Observatories with DISCOPAR	77
5.1	Creating a New Citizen Observatory	78
5.1.1	Mobile App Design Interface	80
5.1.2	Data Processing Design Interface	81
5.1.3	Mobile Data Collection App	83
5.1.4	Observatory Data Analysis Interface	84

5.2	Creating a Campaign	86
5.2.1	Campaign Design Interface	88
5.2.2	Campaign Analysis Interface	90
5.3	Citizen Observatory Meta-Platform Tools	91
5.3.1	Sensor Calibration Tool	92
5.3.2	Community Component Creator	93
5.4	Conclusion	94
6	Implementation	95
6.1	DISCOPAR: Component Layer	96
6.1.1	Components	96
6.1.2	Processes	103
6.1.3	Ports	105
6.1.4	Observations	111
6.1.5	Graphs	112
6.1.6	Distributed Connection Manager	117
6.2	DISCOPAR: Graph Layer	120
6.2.1	DISCOPAR ^{DE}	121
6.3	Citizen Observatory Meta-Platform Architecture	123
6.3.1	Mobile Data Collection App	123
6.3.2	Server-Side Data Processing and Campaigns	126
6.3.3	Web-based Visualisations	126
6.4	Conclusion	127
7	DISCOPAR at Work	129
7.1	Introduction	129
7.2	Citizen Observatory Meta-Platform Expressiveness	130
7.2.1	NoiseTube ^{2.0} : Observatory on Noise Pollution	131
7.2.2	Trage Wegen: Observatory on Pedestrian Experience	139
7.2.3	SensorDrone: Observatory on Atmospheric Conditions	143
7.3	End-User Usability Tests	145
7.3.1	Quasi-Experiment: Mobile App Design Interface	145
7.3.2	Quasi-Experiment: Campaign Design Interface	151
7.4	Conclusion	152
8	Conclusion	155
8.1	Summary	156
8.1.1	Contributions	157

8.2	Future Work	159
8.2.1	Hybrid Mobile Apps	159
8.2.2	Performance	160
8.2.3	User Experience Enhancements	162
8.3	Closing Remarks	164
A	Component List of DISCOPAR	165
A.1	Sensing Components	166
A.2	Logic Components	167
A.2.1	Geographical Components	167
A.2.2	Temporal Components	167
A.2.3	Contextual Components	167
A.2.4	Miscellaneous	168
A.3	Aggregation components	169
A.4	Coordination components	169
A.5	Visualisation components	170
B	Survey Questions	171

LIST OF FIGURES

1.1	DISCOPAR ^{DE} , a web-based visual programming environment for DISCOPAR.	6
2.1	Sensor growth in smartphones.	13
2.2	Architectural overview of a typical participatory sensing system. . .	15
2.3	Campaign lifecycle.	24
2.4	Stakeholders of a citizen observatory.	27
3.1	Citizen observatory architecture.	36
3.2	A simple dataflow program (a) and its directed graph representation (b) [66].	44
3.3	Sample Scratch script [110].	48
3.4	Taxonomy of visual aids for programming [119].	49
3.5	Node-RED's browser-based editor.	52
3.6	FlowHub IDE.	52
3.7	Example of LabVIEW Block diagram.	53
3.8	The Reaktor environment.	54
3.9	The Blender Node Editor.	54
4.1	DISCOPAR ^{DE} : DISCOPAR's web-based visual programming environment.	59
4.2	Overview of DISCOPAR.	60
4.3	Example of DISCOPAR's visual syntax for components.	61
4.4	Visual syntax of connections in DISCOPAR.	62
4.5	DISCOPAR's supported data types and corresponding colours. . . .	63
4.6	The SensorDrone component.	65
4.7	A graph containing distributed components.	66
4.8	DISCOPAR's default visual programming environment.	67
4.9	Menu shown when right-clicking a component on the canvas.	68

4.10	Highlighting of compatible ports.	69
4.11	LineChartVisualisation component configuration window.	70
4.12	Customised component configuration window.	71
4.13	Graph Validation error message example.	72
4.14	Levels of liveness in visual programming systems.	75
5.1	Automatically generated web pages of a citizen observatory.	79
5.2	The Mobile App Design Interface, featuring the component library (left), designer canvas (middle), and a live preview of the mobile app (right).	80
5.3	The Data Processing Design Interface, featuring the component library (left), designer canvas (top-right), and the visualisations preview window (bottom-right).	82
5.4	The Mobile Data Gathering App with the main tab (left) and the campaign tab showing intermediate results (right).	84
5.5	The Observatory Data Analysis Interface.	85
5.6	Automatically generated web pages of a campaign.	87
5.7	The Campaign Design Interface.	89
5.8	The Campaign Analysis Interface.	91
5.9	The Sensor Calibration Tool.	92
5.10	Community Component Creator.	94
6.1	Visual representation of a <code>ObservationData</code> process.	101
6.2	Configuration window of a <code>MapVisualisation</code> process.	102
6.3	Gate ports of a community component.	111
6.4	Simple example of a graph in DISCOPAR.	112
6.5	System architecture of a citizen observatory.	124
7.1	Implementation of NoiseTube ^{2.0} 's mobile app in DISCOPAR.	132
7.2	Radisson Blue Campaign Design.	133
7.3	Geographical constraint editor of Campaign Design Interface.	134
7.4	Visualisations on the 'Radisson Blue' campaign dashboard.	135
7.5	Visualisations on the 'Zayed University' campaign dashboard.	136
7.6	Implementation of the 'Salaam St. and neighbourhood' campaign.	137
7.7	Instructing a participant to the nearest incomplete campaign objective.	138
7.8	Implementation of the mobile app for the Trage Wegen observatory.	140
7.9	Mobile app of the Trage Wegen observatory.	141
7.10	Participant reporting a negative element on the walking trail.	141

7.11	Dashboard of the Trage Wegen campaign in Anderlecht.	142
7.12	Implementation of Sensordrone mobile app.	144
7.13	Sensordrone mobile app and external SensorDrone sensor.	144
7.14	Noise measuring app implementation with redundancies.	147
7.15	Correct, low-level noise measuring app implementation.	148
7.16	Survey results (part 1).	149
7.17	Survey results (part 2).	151
7.18	Campaign created during the “Reclaiming the city” workshop. . . .	152



INTRODUCTION

In 2006, Burke et al. [65] introduced the concept of *participatory sensing* (PS), which tasks individuals, acting alone or in groups, along with their personal smartphones, to systematically monitor personal information (e.g. health) and environmental information (e.g. noise levels, traffic conditions). Given enough people with a smartphone, participatory sensing has the potential to collect enormous volumes of highly localised, person-centric data, which can support (or nudge) policy makers to assess (or adjust) societal or environmental processes in a way that was hitherto unthinkable. Especially in urban contexts with a high concentration of smartphone users, this form of sensing enables the assessment of behavioural and environmental parameters on a scale and a level of granularity that was unattainable before.

The emergence of the PS paradigm has resulted in a broad spectrum of systems targeted at collecting data across various domains. These include, but are not limited to, environmental monitoring, intelligent transportation, personalised medicine, and epidemiological investigations of disease vectors [126]. All these applications are so-called ad-hoc participatory sensing systems, which means that they can only be deployed to collect data about their particular domain, i.e., they are designed on a per use case basis. Developing a new PS system for a particular domain is a costly and time-consuming operation. This inspired several research initiatives to adopt a different approach. Rather than developing yet another ad-hoc PS system, a new generation of platforms [27, 13, 74, 123] has emerged that enable end-users to *configure* their own mobile data gathering apps. Up until now, however, these platforms have limited expressiveness as they focus only on *discrete* data, i.e., single-shot observations which usually take the form of a questionnaire. They provide no support for *continuous* data streams originating from smartphone sensors. This is due to the fact that uploaded surveys are much easier to handle than continuously streaming sensor data.

Despite the inferior quality of smartphone sensors when compared to professional measuring equipment, participatory sensing can still produce qualitative results [130, 41, 64]. This is accomplished by using the potentially enormous quantity of data PS can collect to compensate for the typically inferior quality of individual measurements. But this is easier said than done. To ensure that a high data quality is achieved, PS is currently mainly used on a smaller, coordinated scale where a limited number of people are carefully organised in order to contribute qualitative and useful data. This is achieved by organising a so called participatory sensing *campaign*, which usually focuses the combined data collection effort in both space and time. Scaling up such campaigns in order to truly get quality out of quantity is currently an open problem.

Participatory sensing provides the enabling technology to deploy so-called *citizen observatories*. These are distributed software platforms that provide stakeholders with the instruments to collect, process, analyse, and visualise data in order to accumulate knowledge into a centralised repository. Citizen observatories have been discussed as an increasingly essential tool for better observing, understanding, protecting, and enhancing our environment [84]. Citizen observatories may also serve as an information hub for citizens. For example, a citizen may consult (the data produced and processed by) an observatory on noise pollution to get an idea of the noise pollution that he is exposed to in his daily life. Alternatively, citizen observatories can be a provider of publicly available data that stakeholders can use in their policy making. Various citizen observatories have emerged to establish interaction and

co-participation between citizens and authorities about various environmental issues, emergencies (e.g., flooding [81]), or the day-to-day management of fundamental resources (e.g., FixMyStreet [45]).

1.1 Problem Statement

One major issue of citizen observatories is that, despite the high societal demand, developing a citizen observatory remains a labour-intensive (i.e., costly) and lengthy process that requires substantial technical expertise. Constructing a new citizen observatory for a new type of data (e.g., air pollution, mobility patterns of users of public transportation, etc.) requires most of the software infrastructure to be rebuilt from scratch. Source code for mobile devices, the observatory's website, server-side data processing and data storage infrastructure, participant feedback provision, and data analysis need to be manually programmed time and time again.

Another issue is that even though the concept of a well-orchestrated PS campaign is crucial for ensuring qualitative data, none of the existing PS systems and citizen observatories provide technological support to make sure that the campaign is effectively executed in the foreseen way. Actively orchestrating campaigns is a tedious task even for relatively small campaigns. This is mainly caused by the fact that a significant number of tasks, such as checking the data to verify whether participants satisfied the spatio-temporal constraints imposed by the campaign, still have to be performed manually due to the lack of technological support [36]. The workload involved in manually orchestration campaigns, combined with the lack of technological support for the definition and enactment of a campaign in existing PS systems, are a significant hindrance in the wide-scale adoption of PS campaigns that involve large numbers of participants.

Third, the lack of a systematic, easy, and reusable method for setting up new citizen observatories and for defining new campaigns (within a citizen observatory) poses an insurmountable hurdle for communities and organisations as these typically lack the specific technical ICT-skills and programming knowledge that is needed to create the necessary server infrastructure and mobile applications. This often forces organisations to opt for a non-technological approach (i.e., pen and paper) or to spend large parts of their (often restricted) budget on external ICT-consultants.

Finally, by far the largest obstacle of current approaches is the discrepancy with respect to software engineering efforts: stakeholders lack the strong technical background required to set up their own citizen observatories, and platform engineers can-

not keep up with the demand for (and the number of variations in) citizen observatory and campaigning requirements within limited budgetary constraints.

1.2 Research Vision

What is needed to tackle the aforementioned problems is a generic approach targeting reusable and reconfigurable citizen observatories that are easily accessible by societal stakeholders and communities.

In this dissertation, we focus on finding a solution toward wide-scale adoption of citizen observatories. We propose the notion of a *citizen observatory meta-platform*:

A **citizen observatory meta-platform** is a platform that reasons about and acts upon citizen observatory platforms. It is a platform capable of constructing citizen observatory platforms.

Through this CO meta-platform, citizen observatories can be constructed by stakeholders in a way that appeals to ICT-agnostic end-users. These citizen observatories should be scalable with respect to the amount of data, the number of users, and, most importantly, the type of data collected (i.e., the application area). Only in this way can we move away from small-scale research-oriented deployments to the full-fledged adoption of PS as a societally and scientifically relevant data collection method. The CO meta-platform will exhibit the technical features necessary to concretise the following research visions:

Research Vision 1: Reconfigurable Citizen Observatory Platform The CO meta-platform has to be *generic*, meaning that it must be configurable to provide support for the creation of citizen observatories for any PS scenario. Citizen observatories created through the meta-platform must be expressive and powerful enough to handle both *discrete* data (e.g., people’s experience of their surroundings, such as the perception of local safety in cities) and *continuous* data (e.g., sensorial data, such as temperature or air humidity). Additionally, the underlying technological implementation of each citizen observatory has to be reusable to prevent the re-implementation of each citizen observatory from scratch.

Research Vision 2: Campaign Definition and Enactment Unlike existing PS systems where the definition and enactment of campaigns is not technically supported (and thus managed manually), we envision the CO meta-platform to provide integrated technological support for the concept of a PS campaign. ICT-agnostic users

must be able to define campaigns themselves rather than relying on PS system owners for insider knowledge. Campaigns will be *defined* by means of a set of predicates (e.g., spatio-temporal constraints) about the data collection process that have to be satisfied in order for the campaign to be successfully executed. Each campaign deployed within an observatory will be automatically *enacted* by the platform and provide automated orchestration of participants. This orchestration can coordinate participants while they are collecting data and provide them with feedback on their contributions as an incentive mechanism.

Research Vision 3: Reactive Citizen Observatories In order to guarantee successful campaigning it is essential that we embed automated orchestration to steer participants to optimise their data collection efforts and to provide them with immediate user feedback. In order to provide real-time feedback, citizen observatories and campaigns cannot rely on the traditional mechanism such as batch processing of data and query-based data analysis. We therefore envision each citizen observatory created through the meta-platform as a massive cloud-based *reactive* [11] application that reacts on data coming from mobile devices, contributes those data to the server, and promptly pushes feedback such as intermediate campaign results back to the relevant devices.

Research Vision 4: ICT-Agnostic Usability In order to ensure ICT-agnostic usability of the CO meta-platform, we envision *end-user-centric tools* for constructing citizen observatories and defining campaigns. These tools offer the means to describe the type of data that is of interest, the constraints it is subject to, the mechanism of its collection method, and the process of its storage in the citizen observatory. Through these tools we increase the usability of the CO meta-platform by avoiding the technological complexity of setting up the necessary infrastructure.

1.3 Methodology

In order to concretise the envisioned characteristics of a citizen observatory meta-platform, this dissertation introduces DISCOPAR, a new visual reactive flow-based domain-specific language. DISCOPAR is designed specifically to hide the non-essential complexity of creating citizen observatories and their inherent distributed nature from the end-user, and to present only concepts that are relevant to their domain. DISCOPAR is used throughout the meta-platform to construct every part of a

citizen observatory, i.e., the mobile data gathering app, the server-side data processing, and the web-based visualisations can all be set up using a single visual language.

Furthermore, in order to enable ICT-agnostic users to construct their own citizen observatories and define their own campaigns, we have developed DISCOPAR^{DE}, a web-based visual programming environment for DISCOPAR. A sneak preview of DISCOPAR^{DE} is depicted in fig. 1.1. Programs in DISCOPAR can be created by selecting components from the component menu, and visually assemble them on a canvas through drag-and-drop interactions.

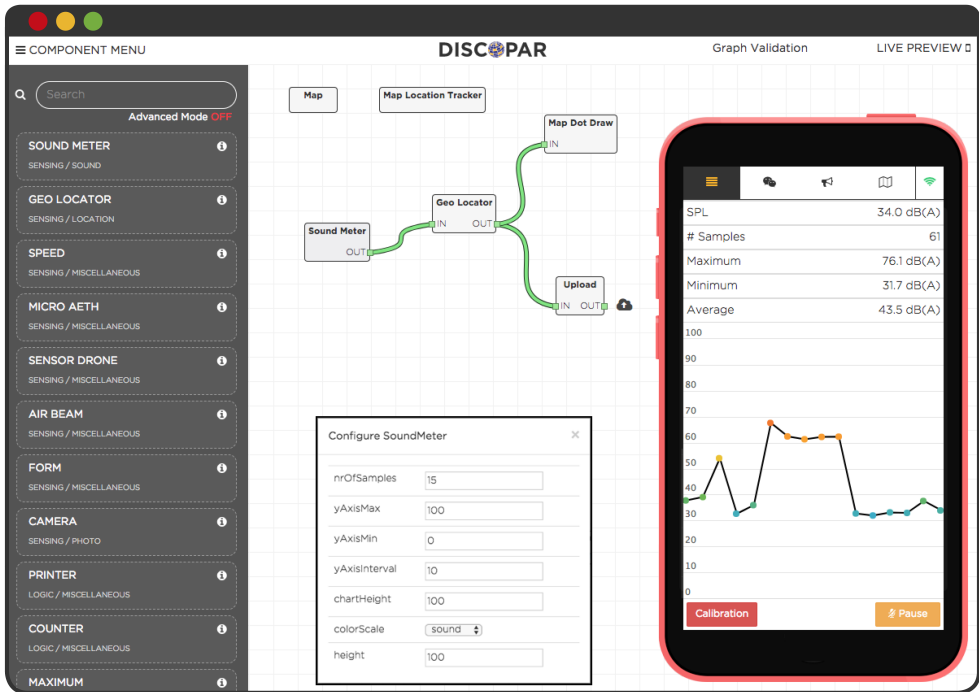


Figure 1.1: DISCOPAR^{DE}, a web-based visual programming environment for DISCOPAR.

1.4 Supporting Publications and Technical Contributions

Several parts of this dissertation's contributions have been published. This section lists these publications and briefly highlights their relevance to this work.

Published Papers

The concept of a participatory sensing campaign is formally introduced in the following papers:

- Ellie D’Hondt, Jesse Zaman, Eline Philips, Elisa Gonzalez Boix, and Wolfgang De Meuter. Orchestration support for participatory sensing campaigns. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp ’14, pages 727-738, New York, NY, USA, 2014. ACM. doi: 10.1145/2632048.2632105. URL <http://doi.acm.org/10.1145/2632048.2632105>
- Jesse Zaman, Ellie D’Hondt, Elisa G. Boix, Eline Philips, Kennedy Kambona and Wolfgang De Meuter, Citizen-friendly participatory campaign support. in *IEEE International Conference on Pervasive Computing and Communication Workshops*, pages 232-235, Budapest, Hungary, 2014. IEEE. ISBN 978-1-4799-2736-4. doi: 10.1109/PerComW.2014.6815208. URL <http://ieeexplore.ieee.org/document/6815208/>

The design of the mobile application builder is presented in the following work:

- Jesse Zaman, Lode Hoste, and Wolfgang De Meuter. A flow-based programming framework for mobile App development. In *Proceedings of the 3rd International Workshop on Programming for Mobile and Touch*, PROMOTO ’15, pages 9-12 , New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3908-7. doi: 10.1145/2824823.2824825. URL <http://doi.acm.org/10.1145/2824823.2824825>

The DISCOPAR platform is discussed in the following paper:

- Jesse Zaman and Wolfgang De Meuter, DISCOPAR: Distributed components for participatory campaigning, *IEEE International Conference on Pervasive Computing and Communication Workshops*, pages 160-165 St. Louis, MO, USA, 2015. IEEE. ISBN 978-1-4799-8425-1. doi: 10.1109/PERCOMW.2015.7134012. URL <http://ieeexplore.ieee.org/document/7134012/>

The underlying distribution model is the focus of the following paper:

- Jesse Zaman and Wolfgang De Meuter, Crowd Sensing Applications: A Distributed Flow-Based Programming Approach, *IEEE International Conference on Mobile Services*, MS '16, pages 79-86, San Francisco, CA, USA, 2016, IEEE. ISBN 978-1-4799-2736-4. doi: 10.1109/MobServ.2016.22. URL <http://ieeexplore.ieee.org/document/7787058/>

Technical Contribution

The DISCOPAR platform was designed as a generic citizen observatory creation tool. It is available online :

- Jesse Zaman, DISCOPAR Platform. <https://discopar.net/>, 2018

1.5 Dissertation Outline

The rest of this dissertation is organised as follows.

Chapter 2: Participatory Campaigning and Citizen Observatories provides a gradual introduction to the core concepts of participatory sensing, participatory campaigning, and citizen observatories. First, an overview of the history of PS is presented. Then, we introduce a definition for a PS campaign, along with a life-cycle model that most campaigns adhere to. Next, we present the concept of a citizen observatory and its relation to participatory campaigning. We discuss a citizen observatory's stakeholders and provide an overview of existing citizen observatory platforms. We highlight the problems involved in creating a citizen observatory. These motivate the need for a citizen observatory meta-platform, i.e., a platform that can create citizen observatory platforms.

Chapter 3: Towards a Citizen Observatory Meta-Platform: Requirements provides a more in-depth analysis of the challenges involved in creating a CO meta-platform. By analysing the “typical” citizen observatory architecture and stakeholders involved, we identify five key requirements that a CO meta-platform has to take into account. Next, we motivate our choice of programming techniques that will be employed to implement these requirements, and we argue that a visual reactive flow-based domain-specific language is the most suitable approach to implement a citizen observatory meta-platform. This chapter therefore introduces the concepts of flow-based programming, reactive programming, visual programming languages, and domain-specific languages, and ends by presenting related work.

Chapter 4: Language Concepts of DISCOPAR focusses on DISCOPAR’s graph layer. More specifically, it presents the visual programming language that is built on top of DISCOPAR’s component layer. We first describe the general idea of DISCOPAR and introduce DISCOPAR^{DE}, our web-based visual programming environment for designing programs in DISCOPAR. Next, we introduce the various concepts and visual syntax of DISCOPAR. We end the chapter by introducing four common strategies of visual programming languages that facilitate end-user programming, and describe how these strategies are applied to DISCOPAR and DISCOPAR^{DE}.

Chapter 5: Constructing Citizen Observatories with DISCOPAR introduces the CO meta-platform. The first part of this chapter focusses on how a citizen observatory can be created through the use of DISCOPAR^{DE}. We describe how the mobile data collecting app, server-side data processing, and web-based visualisations of a citizen observatory can all be implemented in DISCOPAR. The second part of this chapter explains how campaigns can be created within a specific citizen observatory. We conclude the chapter by describing the built-in calibration tool and the community component creator provided by the CO meta-platform.

Chapter 6: Implementation gives an overview of the implementation of DISCOPAR for the sake of reproducibility. This chapter first introduces the various abstractions that together define the component layer in DISCOPAR. Next, we introduce the implementation of DISCOPAR’s graph layer and DISCOPAR^{DE}. Finally, we provide a high-level view on the underlying system architecture of the citizen observatory meta-platform and discuss the various technologies involved in every part of a CO’s architecture.

Chapter 7: DISCOPAR at Work validates our reusable and reconfigurable approach for ICT-agnostic users to design and deploy citizen observatories. The CO meta-platform is validated in terms of expressiveness, suitability, and usability through experiments in both laboratory as well as real-world conditions.

Chapter 8: Conclusion summarises the advantages of our CO meta-platform, provides an overview of the contributions of this dissertation, and highlights some directions for future work.

2

PARTICIPATORY CAMPAIGNING AND CITIZEN OBSERVATORIES

The evolution of the smartphone as a computing platform, combined with the rich sensorial abilities it has acquired in recent years, have led to a new data gathering paradigm called participatory sensing. Participatory sensing (PS) is often used in so called campaigns, i.e., a collective effort focussed in an area and/or time. Participatory sensing is the driving technology behind so-called citizen observatories; i.e., distributed software platforms that provide stakeholders with the instruments to collect, process, analyse, and visualise data in order to accumulate knowledge into a centralised repository. Today citizen observatories have to be developed from scratch for each application area, meaning that deploying a new citizen observatory remains extremely difficult and labour-intensive. Despite an overwhelming demand for such platforms, they are thus beyond the reach of most societal stakeholders. This led us to the idea of creating a reconfigurable citizen observatory platform that allows end-users to build and configure their own citizen observatory.

This chapter provides a gradual introduction into the core concepts of participatory sensing, participatory campaigning, and citizen observatories. First, an overview of the history of PS is presented. Then, we introduce a definition for a PS campaign, along with a life cycle model that most campaigns adhere to. Next, we present the concept of a citizen observatory and its relation to participatory campaigning. We discuss a citizen observatory's stakeholders and provide an overview of existing citizen observatory platforms. We highlight the problems involved in creating a citizen observatory. These motivate the need for a *citizen observatory meta-platform*; a platform that can create citizen observatory platforms.

2.1 Sensors Galore

Conventional data gathering methods used by governments and authorities (i.e., using a limited number of professional sensors placed in key locations) have some limitations [89, 35, 113]:

1. The number of professional sensors deployed is usually limited. Data collection at sparse locations does not scale to meet a high spacial and temporal granularity.
2. Deploying a large number of professional sensors is expensive, discouraging governments and authorities from applying this method due to budget limitations.
3. Professional sensors are usually placed in fixed locations, which means they are inherently measuring the environmental conditions at given places and not those surrounding actual people.

Paulos. et al [42] give a good example of this last problem: *“The civic government may say that the temperature is currently 23°C by taking one measurement at the center of the city or averaging several values from multiple sites across town. But what if you are in the shade by the wind swept waterfront where it is actually 17°C or waiting underground for the subway where it is a muggy 33°C.”*

Faced with these issues of conventional data gathering methods, scientists and stakeholders noticed the technological advances of modern mobile phones and the sheer number of them in circulation. According to a survey [19] conducted in 40 nations, smartphone ownership stands at 43% of the population. The ownership ratio is even significantly higher among the richer economies surveyed (e.g., 88% of South Koreans own a smartphone). The popularity of smartphones is not surprising as these omnipresent devices are much more than mere cell phones that enable users to communicate. They can be regarded as pocket-sized computers that can be used as a phone, MP3 player, point-and-shoot digital camera, hand-held gaming system, GPS, flashlight, alarm clock, e-reader, voice recorder, etc. They have the capability to connect to the Internet, and utilise an operating system capable of running downloaded apps.

As smartphones matured as computing devices, they were gradually equipped with an increasing number of sensors. For example, Figure 2.1 depicts the growth of sensors in the Samsung Galaxy S smartphone series. Nowadays, smartphones are equipped with a gyroscope, compass, accelerometer, proximity sensor, ambient light

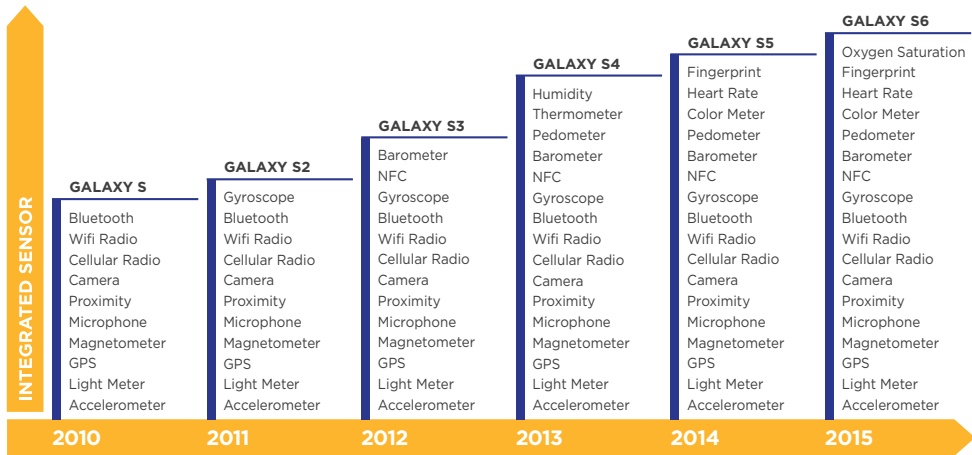


Figure 2.1: Sensor growth in smartphones.

sensor, and other more conventional sensors such as front and back facing cameras, a microphone, GPS, WiFi, and Bluetooth radios. Many of the newer sensors are added to support the user interface or augment location-based services [80]. For example, a light sensor allows the phone's software to automatically adjust the display's brightness, while the GPS enables location-based applications such as maps. However, all these sensors have additional potential: they enhance our ability to measure the real world around us while we carry out our normal daily activities. For example, they can provide real-time information about the current temperature, or we can use them to determine whether noise levels fall within certain limits.

The use of smartphones as sensor nodes and location-aware data collection instruments led to the establishment of a new data gathering methodology, referred to as *participatory sensing* [65, 42]. This approach to data collection and interpretation relies on individuals, acting alone or in groups, along with their personal smartphones to systematically monitor personal information (e.g. health) and/or environmental information (e.g. noise levels, traffic conditions). Given enough people with a smartphone, participatory sensing has the potential to collect enormous volumes of highly localised, person-centric data, which can support (or nudge) policy makers to assess societal processes in a way that was previously unthinkable.

2.2 Participatory Sensing

Participatory sensing (PS) allows people-centric environmental monitoring through the use of smart mobile devices. This is a well-established research field, as witnessed by the publication of recent surveys [32, 126]. The concept of PS is also sometimes referred to as *mobile crowdsensing* [49], *participatory urbanism* [105], *citizen sensing* [65], *urban sensing* [15], or *community sensing* [76].

A concept closely related to participatory sensing is the concept of citizen science. Citizen science is defined by the European Commission as [24]:

“The engagement of the general public in scientific research activities where citizens actively contribute to science either with their intellectual effort or surrounding knowledge or with their tools and resources.”

While participants in citizen science projects provide experimental data and facilities for researchers, they also raise new questions and co-create a new scientific culture. Through citizen science activities, volunteers acquire new learning, skills, and a deeper understanding of the scientific work in an appealing way.

Strictly speaking, citizen science differs from PS in the sense that the former does not necessarily involve the use of technological equipment (smart mobile devices, dedicated sensors, etc.) to collect, store, and share observations from volunteers. For example, the earliest citizen science project is probably the Christmas Bird Count that has been run by the National Audubon Society in the USA every year since 1900. The project is still ongoing, and in the most recent count, tens of thousands of observers counted a total of over 63 million birds [117].

As the name implies, the collection of measurements is done *participatively*, a term used in contrast with *opportunistic* sensing systems. These terms refer to what roles people, as sensing device custodians, are willing to play in large-scale sensing systems [78]. In the case of participatory sensing, people are incorporated into all decision stages of the sensing system, such as deciding what data is shared or when data is gathered. Hence, a PS system focuses on tools and mechanisms that assist people to share, publish, search, and interpret the information collected. With opportunistic sensing, the user may not be aware of applications running in the background. These applications measure data automatically and when appropriate, and can make decisions on their own.

Figure 2.2 presents an architectural overview of a typical PS system. A PS system operates in a centralised fashion, i.e., measurements collected by the mobile phones are uploaded (using wireless data communications) to a central server for processing. On the server, the uploaded measurements are analysed and made available on a

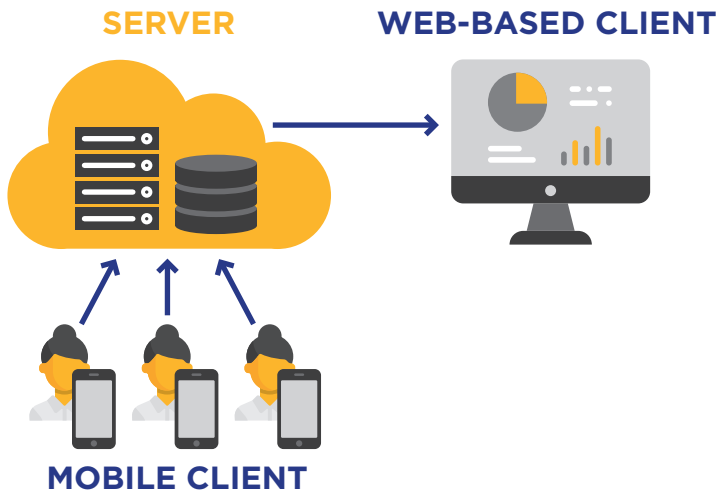


Figure 2.2: Architectural overview of a typical participatory sensing system.

web-portal in various forms, such as graphical representations or maps showing the sensing results.

In the remainder of this section, we provide an overview of related work in the domain of participatory sensing, dividing the research into three categories. The first category consists of PS systems specifically designed for a particular use-case (e.g. measuring noise pollution, monitoring eating habits, etc.). The second category comprises PS platforms that are more general and provide reusable tools that can be shared across various PS projects. The third category focuses on key research challenges related to PS such as privacy, energy, and efficiency.

2.2.1 Ad-Hoc Participatory Sensing Systems

The emergence of the PS paradigm has resulted in a broad spectrum of systems across various domains. These include, but are not limited to, environmental monitoring, intelligent transportation, personalised medicine, and epidemiological investigations of disease vectors [126]. Many of these applications are closed systems in the sense that they can only be deployed in their respective domain, i.e., they are designed on a per use-case basis. We refer to such PS applications as *ad-hoc participatory sensing systems*. We use the term system, as participatory sensing entails more than just a mobile data gathering app. A PS system also includes some data management strategies to capture, analyse, and survey the gathered data, which is usually hosted on some back-end server infrastructure (cfr. fig. 2.2).

We present some examples of ad-hoc PS systems using a similar sub-classification as Christin et al. [32], which is based on the type of sensing involved. The first examples are *people-centric* sensing systems, which mainly focus on documenting activities (e.g., sport experiences) and understanding the behaviour of individuals (e.g., eating disorders). In contrast, *environment-centric* sensing systems collect environmental parameters (e.g., air quality or noise pollution). There exist many more ad-hoc PS systems than those presented in the remainder of this section. We merely use a variety of examples to illustrate the different domains PS can be applied to, in addition to the opportunities that it presents.

People-Centric Sensing Systems

People-centric PS systems collect data about the physiological state and health of their users. With the help of such applications, participants can monitor and document health-related issues.

ExposureSense [106] monitors people's daily activities to compute a reasonable estimation of air pollution exposure in their daily life. ExposureSense further enriches air quality data by combining external sensor network data (e.g., air quality sensors placed on public transports like buses and trams) with data obtained from integrative USB pluggable sensors for smartphones (e.g., a pluggable ozone (O₃) sensor). The daily activities are extracted using the accelerometer sensor as it is suitable for activity recognition with minimal battery consumption.

BeWell+ [79] continuously monitors a user's behaviour along three distinct health dimensions, namely sleep, physical activity, and social interaction. This is done using the gyroscope, accelerometer, microphone, camera, and digital compass. The app promotes improved behavioural patterns by using an ambient display on the smartphone's wallpaper as a feedback mechanism.

StressSense [86] uses the microphone of smartphones to recognise stress from human voice. It does this by analysing a variety of acoustic features of the captured audio samples, such as pitch. The classifier used by the StressSense system can robustly recognise stress among multiple individuals in diverse acoustic environments.

There are many more people-centric applications dealing with health and fitness data such as diet behavior [4, 33, 109], physical activities [121, 57], depression [121], and sport experiences [39, 40]. However, most of these applications have been conceived as research prototypes and their real-world deployment still remains limited in terms of either number of participants or deployment duration [20]. Nevertheless, similar commercial products for documenting running activities, such as RunKeeper [44] and Nike+ Run Club [101], are becoming increasingly popular.

Environment-Centric Sensing Systems

Environment-centric PS systems handle information about a participant's surrounding (e.g., noise pollution, road and traffic conditions, etc.). Unlike most people-centric sensing scenarios, the collected data is mainly aggregated and exploited at a community scale [32].

NoiseTube [89], and similar projects such as NoiseSpy [70], Ear-Phone [107], and Laermometer [9], use the microphones embedded in smartphones to measure the surrounding sound pressure level. These PS systems use their data to build representative noise pollution maps. Sound samples recorded through a mobile phone's microphone can also be analysed to determine the context of the sound recording (e.g., a human voice, music, etc.), as is the case in the SoundSense project [87].

Other projects focus on gathering (urban) air pollution information. Pollution-Spy [69] uses a Bluetooth personal network to connect up to seven different Bluetooth devices (e.g., pollution sensors for CO, NO, NO₂, CO₂, etc.). These sensors feed geolocalised data to a log file and display this collected data graphically on the phone's screen. Users also have the option to transfer the data to a remote database and view it in real time on GIS mapping tools, which are embedded in a dedicated web interface. Similarly, GasMobile[55] uses low-cost sensors connected to smartphones to measure air pollution levels. They provide high data accuracy by exploiting sensor readings near static measurement stations to regularly keep sensor calibration up to date. P-Sense [100] uses external sensors to monitor air pollution and assess the user's exposure to air pollution according to the places visited during daily activities.

Smartphones have also been exploited to document road and traffic conditions. For example, the embedded accelerometer, microphone, and positioning system can be used to monitor traffic and road conditions such as potholes, bumps, or braking and honking, which both are implicit indicators of traffic congestion [92]. In addition, applications such as SeeClickFix [114, 90] and FixMyStreet [45] collect participant feedback about neighbourhood issues (potholes, graffiti, damaged infrastructure). The reports made by users can then, for example, be submitted to the city council and made publicly available. This makes it easy to see what the common problems are in a given area, and how quickly they are fixed.

2.2.2 Reusable Participatory Sensing Systems

Developing a new PS system for a particular domain is a costly operation, as it requires an enormous development effort to implement mobile apps, web interfaces, databases, data analysis, and data visualisation from scratch. This inspired a couple of research initiatives to adopt a different approach. Rather than developing an ad-

hoc PS system, platforms were created that enables end-users to *configure* their own mobile data gathering apps. However, these platforms focus only on *discrete* data, i.e., single-shot observations usually collected through digital questionnaires. They provide no support for *continuous* data streams originating from smartphone sensors. This is due to the fact that surveys are much easier to handle than continuous sensor data streams. Examples of these platforms include Epicollect, ODK, SENR, and ohmage.

EpiCollect [27] is a platform designed to provide a simple and intuitive method to set up data collection projects. A web-based form builder enables users to define a single survey to be used for data collection, which can be deployed as both an Android and iOS mobile app. The survey questions can be organised using conditional branching. A rich set of data types is supported. Collected data can be analysed using a web-based data visualiser. Due to the fact that many data collection projects require considerably more complexity than a single form, an enhanced version of EpiCollect (EpiCollect+) has been developed that provides the ability to produce more complex forms with more features [28].

ODK [13] is a modular toolkit that helps semi-professional users to build mobile data collection solutions. ODK has been deployed in many countries, both in the public health domain and environmental monitoring. It started out as a tool for collecting surveys using mobile devices, and although integrated sensor support was under development, to the best of our knowledge this was never released for production. However, it does provide a modular set of tools that helps simplify sensing application development by creating a single interface to connect to both external and built-in sensors.

SENSR [74] is an authoring tool that enables non-programmers to create, share and manage a citizen science project. SENSR combines a visual drag-and-drop programming environment with a mobile application in which people with limited technical expertise can build mobile data collection tools and manage data collectively. SENSR only supports a limited set of data types and provides no access to smartphone sensors besides the GPS.

Ohmage [123] is a modular PS platform that gathers, analyses, and visualises data from both form-based surveys and continuous data streams. It has been used primarily for research in the education and medical fields. Although ohmage does not explicitly have a sensor abstraction framework like ODK, it can consume collected data streams, such as location traces and audio samples, through its application programming interface.

A different approach to reduce the workload involved in constructing participatory sensing systems is offered by AWARE [43]. Rather than providing a platform that is configurable by non-programmers, AWARE provides an open-source mobile instrumentation toolkit whose aim is to reduce the development effort involved in building participatory sensing systems. To do so, AWARE encapsulates implementation details of sensor data retrieval and exposes the sensors as higher-level abstractions. AWARE is available as a library that can be added to any Android development application or plugin using Android Studio's Gradle mechanism.

2.2.3 PS Research Challenges

A variety of research focuses purely on key research challenges that both ad-hoc PS systems and reusable PS systems are faced with. One key challenge in PS is ensuring that the collected data is *representative* for the monitored area. To ensure spatio-temporal density, participatory sensing systems need enough participants and/or a coordination mechanism to assist the data collection process. This can be done by using game elements on location-based services for directly improving the quality rather than quantity [71, 111], minimising the number of participants necessary to satisfy a predefined coverage constraint [139], and using triggers based on spatial models of previously collected data [83].

A participant coordination approach generally requires that the location of each participating device is known by the central coordination mechanism (i.e., a server). This poses a problem for smartphones, as sending frequent location updates to the server consumes battery power. *Energy consumption* is thus another key research challenge, as users are not willing to use PS applications if they drain their batteries considerably faster [128]. For this reason, research also focusses on optimising battery usage of PS systems. One example is the STREET framework [73], which is assisted by a simple localisation scheme during the data collection process that minimises the usage of the GPS sensor.

Another key challenge is *incentive mechanisms*. The quality of the data gathered by PS systems relies on the willingness of mobile users to participate in the collection and reporting of data. Without adequate incentive mechanisms most users are not willing to participate (on a longer run). Therefore, appropriate incentive mechanisms must be designed if participants do not obtain a direct benefit from their contribution. Such mechanisms can either be monetary (e.g., micro-payments [108, 82, 99, 50]) or non-monetary incentives (e.g., gamification [50, 127]). Additionally, if users feel that their *privacy* might be endangered, it is likely that they will be reluctant to par-

ticipate. A number of projects therefore focus on privacy protection in participatory sensing [26, 51, 12].

2.3 Participatory Campaigning

We introduced the concept of participatory sensing as an alternative data gathering methodology to monitor personal and/or environmental information. One question that remains is whether the potentially enormous quantity of PS data can really compensate for the typically inferior quality of individual measurements coming from smartphone sensors. Several research efforts have answered that *quality-quantity* question in the affirmative [130, 41, 64]. But in order to do this, one first and foremost needs to ensure that a high data quantity is achieved. However, participatory sensing is a relatively new paradigm, so the actual number of people that may be interested and capable of participating is currently insufficient in order to obtain qualitative data. As a result, PS is currently mainly used on a smaller, coordinated scale where a limited number of people are still capable of obtaining qualitative and useful data. This is achieved by organising a so called participatory sensing *campaign*, which usually focuses the combined effort in an area and/or time. The idea of organising participatory sensing actions into campaigns was mentioned as early as 2006 by Burke et al. [65]:

“With the right tools, professionals and community groups alike could employ participatory sensing campaigns to gather data about short-term concerns [...] without waiting for a formal project or grant funding— yielding bottom-up, grassroots sensing.”

The idea of a campaign also emerges naturally in the context of participatory sensing as there is high demand by communities of all sorts — grassroots, institution-led, or research-based — to be able to translate local concerns into actual socio-economic campaigns for tackling them.

PS campaigns ensure data density in absence of a large crowd contributing data. This, in turn, is essential for adequate assessment of the concern at hand. Indeed, in a participatory context one has to strike the right balance between quality and quantity of data, statistically averaging over large datasets so as to minimise random errors while at the same time increasing representativeness of values obtained. By focusing measurement efforts in terms of geographical and temporal boundaries it can be ensured that a dense enough dataset is collected.

While data density plays a significant role in the success of PS campaigns, trustworthiness of the acquired data also plays a role. The fact that anyone is allowed to

contribute data exposes the campaigns to erroneous and malicious contributions [68]. Erroneous contributions may be the result of a user unintentionally positioning the smartphone such that incorrect measurements are recorded, e.g., placing the device in a bag whilst recording data for a noise mapping campaign. Malicious contributions may originate from users that deliberately pollute sensor data for their own benefits, e.g., a real estate leasing agent may intentionally contribute fabricated low noise readings to promote his properties in a particular suburb. A campaign thus also needs a certain level of confidence in the contributions uploaded by volunteers, such that the campaign outcome will be useful for the community involved.

Despite the existence of several PS systems and their application in specific campaigns, there is currently no formal notion of a campaign. However, Burke et al. [65] informally defined a campaign as

“geographically and temporally constrained series of systematic operations to gather a particular type of data — using an already-deployed (but not at all static) network of mobile devices”

In the remainder of this section, we introduce the concept of a campaign protocol, which is used to define the concept of a PS campaign. We also discuss the typical lifecycle of a PS campaign. These concepts are described based on our findings from previous work [36, 137], where we analyse a number of participatory sensing campaigns to observe commonalities and detect patterns.

2.3.1 Campaign Protocol

Participatory sensing is an inherently *active* data gathering method; measurements are gathered consciously by users with specific devices. It is thus possible to describe the *context* of the collected data, i.e., information describing the user (e.g., age, gender) and the particular device (e.g., brand, model) used to collect data. We denote this set of context information by $\{C\}$. Data is gathered in a collection of measurements $\{M\}$. An individual *measurement* is a tuple of data points $M = (s_1, \dots, s_n)$ where s_i are sensor readings. We use the liberal meaning of the word *sensor* here, encompassing real sensor data, user inputs, data ingested from existing datasets, or even higher-order data such as that derived by activity recognition algorithms. We write $\{C\} \Rightarrow \{M\}$ to indicate that the context $\{C\}$ contributed dataset $\{M\}$. We then have the following definition.

Definition 1. A campaign protocol is a set of predicates $P_C + P_M$ over context C and measurements M respectively.

Contextual Predicate

The contextual predicate P_C captures the desired context of the data collection. For example, it can be used to indicate what types of users are allowed to contribute and/or what types of devices must be used to perform the data gathering. The ability to express the former is important as our previous experiences with participatory campaigning indicated that there are certain campaigns that only allow trusted individuals to participate, while other campaigns allowed any participant to contribute. The ability to express what types of devices must be used to perform the data gathering is important as restricting the data collection to certain device models, e.g., those with more accurate sensors or a matching calibration profile, can increase data quality.

Measurement Predicate

The measurement predicate P_M generally includes a temporal predicate P_T specifying *when* participants must be collecting data. For example, a temporal predicate can be used when a comparison between days (e.g., workweek vs. weekend) or hours (e.g., peak-hours vs. off-peak hours) is desirable. Temporal predicates can also be used when a limited number of participants is available.

The measurement predicate also frequently includes a geographical predicate, which we write as P_A , with A for the area. Placing a geographical boundary on the area of interest of a campaign has multiple purposes [137]. First, campaigns become ‘local’, i.e., they tackle a concern in a particular area. Because, people living in that area would benefit from participating in that campaign, localising a campaign may serve as an incentive to take part in it.

Focussing participant contributions to a certain time frame and instructing them to measure in a predefined area increases the density of the collected measurements, thereby enabling the generation of more accurate results.

Example: NoiseTube

NoiseTube [89] is a participatory monitoring and mapping system of ambient sound levels through mobile phones. In NoiseTube measurements are tuples of sensor readings containing the time, sound pressure level, latitude, longitude, and zero or more tags that can be used to describe the sound source. We can thus say $\text{userAccount1, SamsungS8} \Rightarrow \{\text{time}_i, \text{Leq}_i, \text{latitude}_i, \text{longitude}_i, \text{tags}_i^*\}_{i=0}^n$ to indicate that that a particular user and mobile device contributed a dataset containing NoiseTube sensor readings. Consider a campaign with the following scenario: a city administrator wishes to investigate sound pressure levels

recorded by his employees, using calibrated handsets, in the centre of Brussels between 8-9am during the month of May. We can thus formally represent the campaign protocol predicates as follows:

$$\begin{aligned}
 P_C &= (\text{user} \in \text{employees}, \text{device} \in \text{calibrated-devices}) \\
 P_M &= ((\text{latitude}, \text{longitude}) \in \text{polygon}(\text{Brussels}), \\
 &\quad \text{time} \in [2017-05-xT08:00, 2017-05-xT09:00] \\
 &\quad \text{with } x \in \{1, \dots, 31\})
 \end{aligned}$$

2.3.2 Campaign Definition

Using the concept of a campaign protocol, we now define a participatory sensing campaign as follows:

Definition 2. *A campaign is a collection of measurements M constrained by the campaign protocol consisting of predicates $P_C + P_M$ over context C and measurements M respectively, such that M is collected participatively by a set of users satisfying P_C . A campaign is considered successful if M is dense enough to generate a qualitative output.*

It is important to notice that in addition to the constraints embodied in the protocol, we also capture the necessity of obtaining data that meets certain qualitative predicates. Whether or not a particular output is qualitative depends on the specific goal of a campaign. The enormous quantity of data that is typical of PS is usually translated into a more qualitative condensed representation, e.g., by location-based statistical averaging. Therefore, the quality of a campaign is inherently related to the density of measurements.

2.3.3 Campaign Lifecycle

From analysing several environmental PS campaigns [136, 36], combined with our extensive expertise in participatory noise sensing with the NoiseTube project, we derived a typical campaign's *lifecycle*, which is shown in fig. 2.3. A campaign's lifecycle proceeds through a number of stages [136, 137]:

Campaign Conceptualisation

The first step in a campaign's lifecycle is the initiative. There are many situations where one or more stakeholders may want to initiate a campaign around a given interest. Examples of campaign *initiators* include:

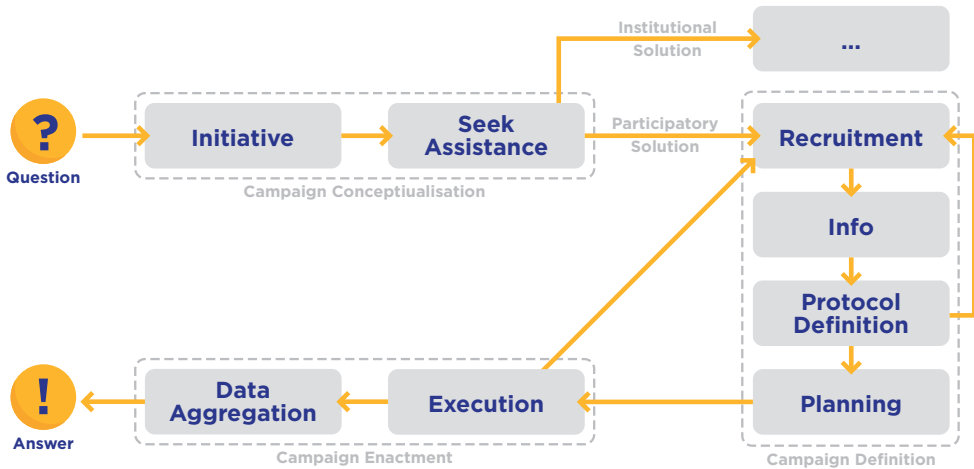


Figure 2.3: Campaign lifecycle.

- A civil society organisation setting up a campaign based on its members organisations’ interests.
- A research institute or governmental organisation setting up a campaign for participatory research or policy-making.
- Citizens setting up a campaign for a particular concern in their neighbourhood.

In second step, the initiators seek a suitable solution for their particular interest and compare existing data gathering methods. During this phase, initiators may discover the existence of a certain PS system either by themselves (media, word of mouth) or by consulting a research institute, governmental organisation, etc. This PS system can then be considered as an alternative to better-known institutional solutions. Once the initiators agree to adopt a participatory solution, the campaign definition starts.

Campaign Definition

Defining a campaign involves several steps: The envisioned participants are recruited and informed about the concept of a PS campaign, and each participant is briefed with the usage instructions of the selected PS system. This is usually done by face-to-face meetings where an expert explains the prerequisites for a campaign to be successful. In case the initiators underestimated the workload involved and the number of participants needed, an additional recruitment phase can occur.

As soon as enough motivated participants have been recruited, the planning phase begins. The purpose of this phase is to ensure the campaign achieves its goal. Remember that for participatory sensing to be a valid alternative for other data gathering methods, a high data quantity must be achieved. The planning ensures that each participant's measurements have optimal use and reduces the chance for data redundancies. For example, a noise mapping campaign concerned about the noise pollution in a certain neighbourhood will set up a planning to prevent that every participant measures in the same area at the same time. Once the planning is complete, the actual execution of the campaign can finally begin.

Campaign Enactment

The actual execution of the campaign consists of participants collecting data using their mobile devices. Ideally, the campaign is monitored and analysed *during* this phase to trigger remediating action when problems arise. For example, when there is insufficient coverage of the area of interest, or when participants are producing unusable data as a result of not measuring properly (e.g., not turning on the GPS of the mobile device).

During the campaign's execution, all the data is aggregated and made available for analysis and visualisations. The campaign's output can then be used by the initiators, participants, and decision makers to reflect upon the results. For example, when people take part in a campaign that keeps track of their travel habits and provides them with personalised recommendations for alternative routes (and benefits related to these alternatives), they may decide to follow up on those recommendations.

2.4 Citizen Observatories

Although many PS systems were implemented as a prototype for a single use-case, there also exist more advanced PS systems that have been used to organise multiple campaigns. In this case, the PS system becomes a sort of data repository for various campaigns, where researchers/citizens — who are not necessarily actively involved — can analyse/compare the data gathered of every campaign, as well as individual contributions.

For example, consider NoiseTube [89], a PS system enabling the monitoring and mapping of ambient sound levels by using the microphone of mobile phones. Between 2008 and 2016, NoiseTube has been used in 19 noise mapping campaigns at 13 different locations. One such example is a group of 7 concerned citizens that mapped the noise pollution in their neighbourhood due to a huge traffic-laden round-

about [36]. In some of these campaigns, the resulting noise maps were used to show the city administrations that existing official noise maps based on simulations were not representative of the actual noise levels. Despite their different intentions, each of these campaigns contributed meaningful data on sound pressure levels to NoiseTube. Combined with standalone contributions of individual users, NoiseTube has become a data repository on noise pollution data from all over the world. End-users are allowed to consult this data for various purposes, such as to compare noise levels in major cities around the world.

To differentiate the single use-case PS systems with these more advanced PS systems that act as a central repository of knowledge on a particular type of data, we refer to the latter as so called *citizen observatories*, which we define as follows:

Definition 3. *A citizen observatory (CO) is a distributed software platform that provides stakeholders with the instruments to collect, process, analyse, and visualise data in order to accumulate knowledge into a centralised repository.*

It is important to understand the relationship between a citizen observatory and the previously introduced concept of participatory campaigning. Generally speaking, a CO collects, analyses, and evaluates a specific type of data (e.g. environmental noise, water pollution, etc.). To collect this data, each observatory has one dedicated mobile PS app. Within each CO, users can deploy campaigns to aggregate data for a particular goal or concern. Campaigns are only interested in a *subset* of the observatory data (defined by the campaign protocol), and thus describe how the observatory data should be filtered, processed, and aggregated. Note that a CO does not necessarily collect data only through campaigns. The idea of a CO is usually to obtain publicly available data that stakeholders can use in their policy making. It is therefore perfectly possible that stakeholders merely use the CO to create visualisations using the available open data without actually running a campaign themselves. Citizen observatories also serve as an information hub for citizens. For example, an individual user might consult the CO data to get an idea of the noise pollution that he is exposed to in his daily life.

The definition of a citizen observatory does not imply that the data collection can only be accomplished using smartphones. As stated by Liu et al. [84] a CO entails “*the participation of citizens in monitoring the quality of the environment they live in, with the help of one or more of the following: (1) mobile devices of everyday utility; (2) specialized static and/or portable environmental and/or wearable health sensors, and (3) personal, subjective and/or objective observations, information, annotation and exchange routes, coming from social media technologies or other similar plat-*

forms”. In the scope of this dissertation, we only focus on citizen observatories where the data collection is performed using smartphones and specialised portable sensors.

2.4.1 Citizen Observatory Stakeholders

By analysing several PS systems, Christin et al. [32] derive a general model of a PS system including stakeholders and architectural components. We extend this model to include the concept of a citizen observatory and the campaigns that it can host. The resulting model is depicted in Fig. 2.4. We now discuss each stakeholder involved in a citizen observatory.

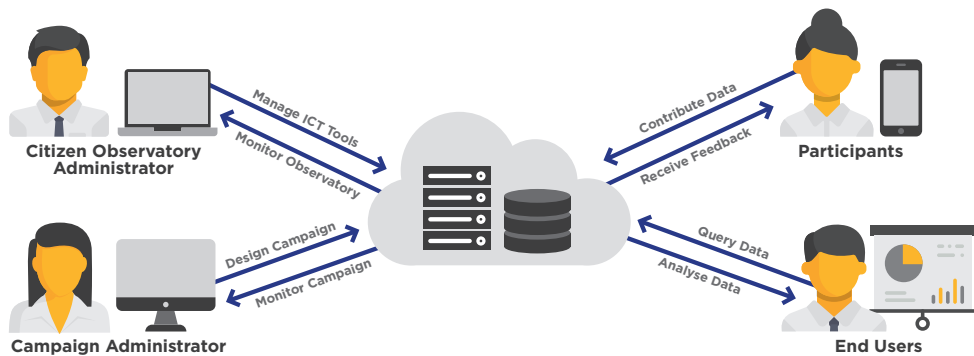


Figure 2.4: Stakeholders of a citizen observatory.

Citizen Observatory administrators are the creators of citizen observatories. They are responsible for creating and managing the set of ICT-tools that comprise a CO, usually a mobile application for data gathering, server-side back-end infrastructure for processing and storing data, and web-based visualisation tools. It is the CO administrator’s responsibility to set up these tools either by themselves or by hiring a developer.

Campaign administrators are members of a community, an organisation, research group, or simply individuals who wish to set up a participatory sensing campaign that uses a certain CO. They design, implement, and deploy the PS campaign within the CO. The campaign is defined as a subset of the data collected by the observatory, so ideally the CO provides campaign administrators the means to describe which subset of data they want to gather and for what particular purpose or goal.

Participants install the PS application on their smartphone and voluntarily gather sensor readings to contribute to the CO and the PS campaigns of that CO. A participant's can be motivated to contribute data either through personal benefit, e.g., to monitor their impact on the environment, or through specialised incentive mechanisms (cfr. section 2.2.3).

End users access and consult the data gathered by the participants according to their interests and preferences. End users include the contributing participants, as they usually want to consult their own collected data. Other end users can be campaign administrators verifying the actual contributions and results, or specialised scientists attempting to gain insights about the monitored phenomena.

2.4.2 Research in Citizen Observatories

Citizen observatories have been discussed as an increasingly essential tool for better observing, understanding, protecting, and enhancing our environment [84]. This gain in popularity is visible, for example, through the many research projects funded by the European Union on the topic of observatories. In 2012, five projects were funded under the EU's Seventh Framework Programme for Research and Technological Development (FP7): CITI-SENSE [22] (air quality), COBWEB [23] (biospheres), WeSenseIt [133] (water quality), Citclops [21] (ocean monitoring), and OMNISCIENTIS [104] (odour monitoring). Each of these projects were aimed at developing novel technologies and applications in the domain of Earth Observation. Their goal was to exploit the capabilities offered by portable devices to enable an effective participation by citizens in environmental stewardship, in support of both community and policy priorities. Ever since, various citizen observatories have been emerging to stablish interaction and co-participation between citizens and authorities about various environmental issues, emergencies (e.g., flooding [81]), or the day-to-day management of fundamental resources.

Research in citizen observatories continues in ongoing projects of the EU's Horizon 2020 research and innovation programme. This programme includes projects such as the GROW observatory [103] that generates, shares, and utilises information on land, soil and water resource, and the SCENT [112] and LandSense [77] observatories that and use and land cover. Citizen observatories such as these prove the critical role of ICT in the evolution towards a sustainable society.

2.4.3 Research Questions

Despite the existing and ongoing research, there are still many fundamental challenges that citizen observatories are faced with, such as ensuring effective user participation, dealing with data privacy, and taking into account data standards, quality and reliability [84]. One major issue of citizen observatories is that, despite the high societal demand, developing a new citizen observatory remains a labour-intensive (i.e., costly) and lengthy process that requires substantial technical expertise. In this dissertation, we focus on finding a solution towards a wide-scale adoption of citizen observatories. Concretely, this dissertation addresses the following research questions:

Research Question 1

How can we transform the development and deployment of new citizen observatories into a more systematic process?

There is currently no systematic, easy and reusable method for setting up new citizen observatories and for defining their corresponding data collection campaigns. Although there already exist several reusable and reconfigurable PS systems (cfr. section 2.2.2), they focus more on data collection (typically form-based) and neglect other roles of a citizen observatory such as advanced data processing methods, and participant feedback and coordination.

One key insight that can be used to make the development and deployment of citizen observatories a more systematic process is that each CO and their underlying participatory sensing applications all share a similar structure: information is gathered collaboratively (implicitly or explicitly) by mobile users and uploaded to a server for aggregation, global analysis, and visualisation. Feedback is sent back to the users so that they are made aware of the status of the observatory, ideally in a continuous manner.

Research Question 2

How can we provide technological support for the definition and enactment of a PS campaign and enable the automated orchestration thereof?

The knowledge on how to organise campaigns properly is absent from currently existing PS systems, nor do they provide any means to define and enact a PS campaign. This is problematic for stakeholders, who are therefore largely dependent on PS system owners for insider knowledge on campaign management, but also for the PS

system owners, who are not able to scale up their efforts in managing these campaigns.

Furthermore, a major factor in the success of PS campaigns (and consequently citizen observatories) is sustained high quality participation. We previously discussed (cfr. section 2.2.3) that incentive mechanism such as gamification and micro-transaction can be used to increase participation. However, as stated by Stevens [120]: “*user behaviour is largely unpredictable and some user actions, or lack thereof, can have detrimental effects on data quality. Usually this happens unintentionally or even unknowingly; for instance due to forgetfulness or a lack of knowledge, skill or time.*” To prevent this unwanted behaviour, it is important that campaigns are orchestrated to provide participants *feedback* on their contributions. Providing direct feedback to participants has multiple benefits [1]:

- Feedback can be used to provide participant coordination on an individual level, ensuring relevant contributions by checking the user’s data to verify whether it satisfies the constraints imposed by the campaign.
- Feedback acts as an incentive for participants to motivate them to increase their contribution or to help them convince others to join, resulting in higher participation and consequently better data quality.

Actively orchestrating campaigns of a citizen observatory to provide such feedback is a tedious task. This is mainly caused by the fact that a significant amount of tasks, such as checking the data to verify whether participants satisfied the constraints imposed by the campaign, still have to be performed manually due to the lack of technological support [36]. The workload involved in manually orchestrating campaigns, combined with the lack of technological support for the definition and enactment of a campaign in existing PS systems, are a significant bottleneck in the wide-scale adoption of PS campaigns.

Research Question 3

How can we step away from the batch-processing philosophy according to which current citizen observatories have been designed and turn the collect-process-analyse-visualise chain into a fully interactive process?

Existing PS systems and citizen observatories are currently developed as *static* applications: collected data is usually processed and analysed through a batch-processing mechanism, after which it is stored in a database. Visualisations are then generated by performing queries on this database.

This traditional approach discourages participants from contributing due to the slow or late feedback, and lack of acknowledgement of their contributions by the citizen observatory and/or campaigns. In contrast, immediate and specific feedback can serve as incentive mechanism for participants. Slow feedback is also problematic for the automated orchestration of a PS campaign (cfr. Research Question 2). For example, steering the movements of individual participants requires real-time knowledge on each participant's whereabouts.

To solve this issue, citizen observatories and campaigns must be responsive to uploads from participants. The data gathering process must be an interactive cycle where participants upload data to a citizen observatory and immediately receive feedback, such as coordination instructions or intermediate campaign results, based on their contributions.

Research Question 4

How do we enable ICT-agnostic stakeholders to design their own citizen observatories and campaigns?

Due to the lack of systematic, easy and reusable method for setting up new citizen observatories, deploying a new citizen observatory remains difficult and labour intensive. They are thus beyond the reach of most stakeholders, who usually lack the necessary ICT-skills and programming knowledge to create a citizen observatory from scratch [18]. This often forces stakeholders to opt for a non-technological approach (i.e., pen and paper) or to spend big chunks of their restricted budget on external ICT-consultants.

To be accessible to societal stakeholders and communities, citizen observatories must be configurable and usable by *ICT-agnostic* stakeholders. Here, the challenge is finding the right balance between expressiveness and usability. Stakeholders must be allowed to design a citizen observatory and campaigns in a way that automatically generates the necessary tools to collect, process, analyse, and visualise data according to their expectations.

2.5 Vision: Citizen Observatory Meta-Platform

Despite the high societal demand, citizen observatory development remains labour-intensive, lengthy, and requires technical expertise. This led us to the idea of a *generic approach towards reusable and reconfigurable citizen observatories*. We refer to this

generic platform for setting up citizen observatories as a *citizen observatory meta-platform*. A ‘meta-platform’ is a platform that reasons about and acts upon another platform [88]. Hence, a CO meta-platform reasons about and acts upon citizen observatory platforms. Concretely, we envision a CO meta-platform as follows:

Research Vision 1: Reconfigurable Citizen Observatory Platform To avoid rebuilding all the mobile apps, web interfaces, databases, data analysis and visualisation elements from scratch, we envision the CO meta-platform as a *component-based* software development approach that is less ad-hoc and less dependent on hardcore programming technologies. Despite being constructed from reusable components, citizen observatories created through the meta-platform must be expressive and powerful enough to handle both *discrete* data (e.g., people’s experience of their surroundings, such as the perception of local safety in cities) and *continuous* data (e.g., sensorial data, such as temperature or air humidity).

Research Vision 2: Campaign Definition and Enactment Unlike existing PS systems where campaigns are not explicitly supported and usually managed manually, we envision the CO meta-platform to enable the definition of a PS campaign and the enactment thereof. As a result, initiators can define campaigns themselves rather than relying on PS system owners for insider knowledge. Campaigns are then enacted by the platform to provide automated orchestration to participants. This orchestration can coordinate participants whilst they are gathering data and provide them feedback on their contributions as an incentive mechanism.

Research Vision 3: Reactive Citizen Observatories To guarantee successful campaigning it is essential that we embed automated orchestration to guide data collection and immediate user feedback. This requires citizen observatories and campaigns to be *reactive*, whereas the automated orchestration must encompass the distributed architecture of a citizen observatory (mobile data gathering app and server-side processing) *transparently*. We therefore envision each citizen observatory created through the meta-platform as a massive cloud-based reactive application that reacts on data coming from mobile devices, contributes that data to the server, and promptly pushes feedback, such as intermediate campaign results, back to the relevant devices.

Research Vision 4: ICT-Agnostic Usability Within this CO meta-platform, stakeholders can set up and configure the necessary ICT-tools of a citizen observatory and deploy campaigns without or with very little programming skills. We

envision an interface that offers access to components to describe the type of data, the constraints it is subject to, the mechanism of its gathering and the process of its storage in the citizen observatory. This interface also enables stakeholders to formulate a PS campaign by defining the campaign protocol (cfr. section 2.3.1) and their expectations for the campaign's output (e.g., types of maps).

Furthermore, the CO meta-platform has the potential to act as a central hub for the various research undertakings in PS. Due to its component-based approach, problems such as incentive or coordination mechanisms (cfr. section 2.2.3) that are usually researched and solved in an isolated environment, can be implemented as a component of the meta-platform. As a result, interoperability and synergies can be established between the various research performed in the domain of participatory sensing.

2.6 Conclusion

In this chapter, we explain how the evolution of the smartphone, combined with its user's mobility, led to the establishment of a new data gathering methodology referred to as participatory sensing. After providing an overview of PS use cases and related research, we introduced the concept of participatory campaigning as a combined data gathering effort and formally described the various aspects of a campaign. In the bigger picture, such campaigns are a part of citizen observatories. We observe from the current state of the art that each of these observatories are developed from scratch, due to a lack of reusable and reconfigurable citizen observatory construction tools. Although there already exist several reusable and reconfigurable PS systems, they focus more on the data gathering (typically form-based) and neglect other roles of a citizen observatory such as advanced data processing methods, and participant feedback and coordination. As a solution, we propose a more generic approach through the use of a CO meta-platform. Through this meta-platform, ICT-agnostic stakeholders will be able to construct their own citizen observatories and design PS campaigns with little or no help from platform owners. Only in this way can we move away from small-scale research-oriented deployments to the full-fledged adoption of PS as a societally and scientifically relevant method. In the following chapter, we perform a requirements analysis of the envisioned CO meta-platform and use these as a guideline to choose the most suited technologies to implement the CO meta-platform.

3

TOWARDS A CITIZEN OBSERVATORY META-PLATFORM: REQUIREMENTS

Building a citizen observatory meta-platform is a challenging task because of three reasons. First, in order to be truly generic, the platform must cover the rich diversity in citizen observatory scenarios. Second, when applicable, campaign participants must be coordinated in real-time to ensure the campaign's data quantity and/or quality. Last but not least, the platform must be accessible to stakeholders and communities with limited ICT knowledge.

This chapter starts by providing a more in-depth analysis of the challenges involved in creating a CO meta-platform. By analysing the typical citizen observatory architecture and stakeholders involved, we identify five key requirements that a CO meta-platform has to take into account. Next, we motivate our choice of programming techniques that will be employed to implement these requirements, where we argue that a visual reactive flow-based domain-specific language is the most suitable approach to implement a citizen observatory meta-platform. Therefore, this chapter introduces the concepts of flow-based programming, reactive programming, visual programming languages, and domain-specific languages. We end the chapter by presenting related work.

3.1 Citizen Observatory Architecture

Citizen observatories all share a similar architecture, namely a mobile app, server with a database, and web-based visualisation tools. Figure 3.1 depicts this typical citizen observatory architecture. This architecture is based on similar reference architectures for participatory sensing systems [138, 37]. We discuss each element in-depth to analyse the functional requirements for a CO meta-platform.

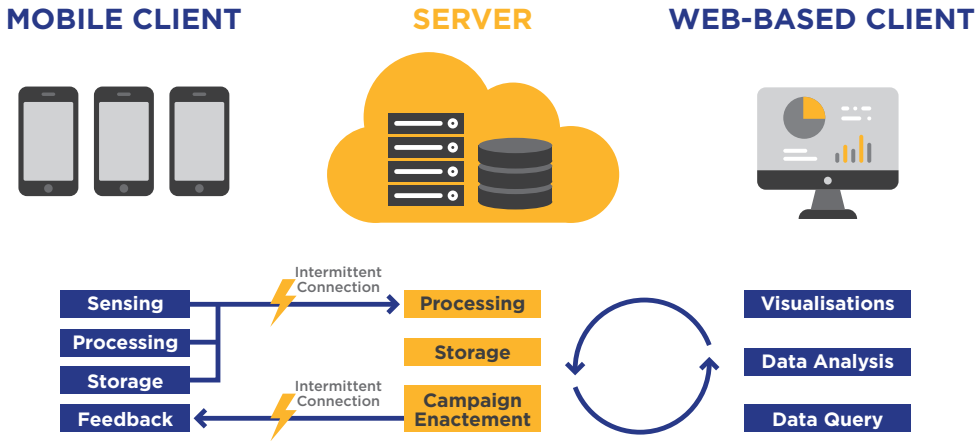


Figure 3.1: Citizen observatory architecture.

Mobile Apps gather the required information through on-board sensors and user-input, and upload it to the server for aggregation and analysis. Additionally, users of the mobile app (i.e., participants) may receive feedback in real-time (e.g., coordination instructions, data visualisations). Despite the high variability of citizen observatory scenarios, PS apps share common technical properties. Regardless of which data parameters are being collected, PS apps need to include logic on how to upload data to a server, handle temporal disconnections, access the device’s sensors, etc.

Servers and Databases are responsible for processing and storing the (potentially massive amount of) data generated by the mobile apps. Similarly to the mobile apps, many different processing steps and algorithms can be used in multiple scenarios (albeit with slightly modified settings and input data). Storing the data also presents a challenge, as each citizen observatory can gather completely different types of data. Additionally, the server has to handle every campaign deployed within the observatory. Each campaign must process and filter uploaded data as specified through the

campaign protocol (cfr. section 2.3.1). Ideally, this campaign enactment occurs in real-time to provide immediate feedback on each participant's contributions.

Web-Based Analysis and Visualisation Tools provide participants and stakeholders means to reason about the collaboratively gathered data. In most cases, these are simple graph visualisations or map-based plotting of data samples. However, more advanced data analysis and visualisations can be envisioned, such as cluster analyses and correlations

3.2 Meta-Platform Stakeholders

As described in section 2.4.1, there are various stakeholders involved in a citizen observatory. The CO meta-platform must be accessible to these stakeholders and communities with limited ICT knowledge. In this section, we provide details on our assumptions on the background and level of ICT knowledge of the various stakeholders that will interact with the CO meta-platform.

CO meta-platform owners are the only stakeholder category that require programming knowledge, as they are responsible for maintaining the CO meta-platform and the corresponding hardware architecture. This includes the addition of new features, such as support for a new type of smartphone sensor.

Citizen Observatory administrators and **Campaign administrators** are expected to be domain-experts with limited ICT knowledge, comparable to the knowledge required for interacting with Microsoft Excel. This means that administrators do not require programming knowledge to set up their own citizen observatory or deploy a campaign using traditional methods. However, administrators do need to be familiar with the domain in the sense that they need to know how to collect, process, and visualise data.

Participants do not need to be domain experts of the citizen observatory that they are contributing to. We assume basic technological knowledge and experience in interacting with a smartphone. This includes the knowledge on how to install and operate a mobile app.

End-Users are divided into two categories: basic end-users and advanced end-users. Basic end-users only look at the visualisations and results made available on the web page of the citizen observatory or campaign. Advanced end-users are those that want to perform their own analysis and create their own visualisations. For advanced end-users, we assume an ICT knowledge-level on par with being capable of working with Microsoft Excel.

3.3 Meta-Platform Requirements

Based on the CO architecture and stakeholder assumptions presented above, and taking into account the research questions (cfr. section 2.4.3) and the concept of a CO meta-platform (cfr. section 2.5), we now introduce the five key requirements for a CO meta-platform: customisability, usability, compatibility, scalability, and reactivity. A distinction is made between requirements relating to the user level and those related to the implementation of the meta-platform. In the remainder of this section, we present the requirements of both categories and discuss which technologies facilitate the development of a truly generic CO meta-platform.

3.3.1 User Level Requirements

The CO meta-platform must enable ICT-agnostic stakeholders to construct their own observatory and campaigns for a variety of scenarios. To enable this, the meta-platform must ensure *customisability* and *usability*. Additionally, participants contributing data to campaigns require real-time feedback. As a result, the CO meta-platform must include support for *reactivity*. We now elaborate on each of these user level requirements (ULR).

ULR-1: Customisability

The meta-platform must enable CO administrators and campaign administrators to set up and customise their CO and campaign respectively, in such a way that it suits their particular scenario. To support multiple scenarios and to be truly generic, the meta-platform must support different types of data and deal with the specific features involved. This includes both *continuous* data (sensorial parameters such as noise, accelerometer, humidity, etc.) as well as *discrete* data (e.g., behavioural parameters obtained from user input through questionnaires). In addition to providing the necessary features to collect such diverse data, the meta-platform must also enable the customisation of different data processing algorithms for each citizen observatory scenario.

Rather than forcing each CO to re-implement the same features with only some minor adjustments, we propose the use of *component-based development model*.

A component-based approach takes advantage of citizen observatories' similar structure to provide end-users with an off-the-shelf toolkit including all relevant components and tools that are needed to set up a citizen observatory. Such a toolkit would, for example, include:

- **Data gathering components** for collecting data either using the smartphone's sensors, or by requesting user input.
- **Data processing components** for processing and aggregating data based on a particular algorithm.
- **Feedback components** for handling interaction with the user (e.g., instructions on how to perform the measurements, feedback coming from the server, gaming elements, etc.).
- **Distribution components** for handling the client-server communication (e.g., upload of measurements to the observatory).

A CO can then be implemented through a composition of such reusable and configurable components, enhancing the customisability of the meta-platform. Therefore, it is desirable that the toolkit contains a component for as many features as possible, and that each component is sufficiently customisable to cope with the variety of CO data types.

ULR-2: Usability

To be accessible to societal stakeholders and communities, the meta-platform must enable the deployment of a citizen observatory and the setup of a campaign without or with very little programming skills. Expecting stakeholders to program a citizen observatory using textual source code is therefore unreasonable. A more accessible and realistic solution is to provide stakeholders with a graphical interface in which they can assemble a CO and campaigns in a way similar to playing with LEGO or Scratch [110]. In fact, the aforementioned component-based development model serves as a good foundation to create a *visual programming language* (VPL). Basic programming elements in a VPL are generally represented as blocks, i.e., structural elements with visual cues about how they can be used and linked together. By only allowing the blocks to be arranged in a way that "fits", incorrect programs can be prevented from being constructed. In contrast, there is a big risk that stakeholders do not manage to program their own citizen observatory textually as, in addition

to the syntax itself being a hindrance, there is no built-in visual cue to prevent the arrangement of components in an incorrect way.

Not any VPL will suffice. It is important that stakeholders can express themselves using familiar concepts without puzzling over implementation technicalities. As a result, we propose to restrict the VPL to a particular domain, namely the domain of participatory sensing. Using this visual *domain-specific language* (DSL), stakeholders can program a CO (and its campaigns) simply by describing the data parameters that must be collected, which data processing and aggregation algorithms must be used, the type of feedback participants receive, etc.

ULR-3: Reactivity

Each participant is connected in real-time to the observatory and, ideally, receives immediate feedback about his/her contributions. This means that every time a mobile device uploads new information, the corresponding citizen observatory must process it immediately, keep track of the progress of the running campaigns, and inform the “interested” mobile devices about the freshly updated state of every campaign. Additionally, each observatory is responsible for coordinating the behaviour of the campaign participants to ensure an optimal data density. Such a coordination mechanism must be capable of operating in real-time, as it makes no sense to send participants instructions if they already moved to a different location or stopped contributing data by closing the app.

Orchestration of an observatory/campaigns involves more than just participant coordination. Participants must receive feedback as incentive mechanisms, campaign creators must be informed about the progress of their campaign, data visualisations must be kept up to date, etc. Ideally, each of these happen *in real-time*. It is therefore important that the meta-platform is capable of immediately *reacting* to data coming from mobile devices, contributing those data to the server and promptly pushing intermediate campaign analysis results back to the relevant devices.

3.3.2 Implementation Level Requirements

A CO meta-platform will host multiple citizen observatories, each hosting several campaigns. The high variety of data types and the different types of hardware involved (e.g., smartphones with different OS) requires the meta-platform to ensure *compatibility* of the various elements in the CO distributed architecture. Additionally, the large amount of participants and end-users simultaneously interacting with the meta-platform requires *scalability*. We now discuss each of these implementation level requirements (ILR) in more detail.

ILR-1: Compatibility

The architecture of a citizen observatory as depicted in fig. 3.1 engenders a *distributed* architecture: mobile apps running on smartphones communicate across a network to upload data (in real-time) to a processing server. Feedback is sent back to these mobile devices and to any web-based clients that are monitoring the results of the processing (e.g., visualisations) on dedicated webpages.

In existing citizen observatories and other PS systems, these various elements of the architecture have been developed using a different set of technologies and programming languages:

- Mobile apps have to be coded for their respective operating system (i.e., Java for Android , Objective-C or Swift for iOS).
- The data processing server requires knowledge on both web server technology (e.g., Apache Tomcat, Ruby-on-Rails, etc.) and data processing techniques (e.g., Spark, etc.).
- A database system has to be chosen (e.g., SQL or some NoSQL solution).
- Building web-based visualisation tools requires expertise in HTML, CSS, and JavaScript.

Constructing a citizen observatory involves creating each element of the architecture shown in fig. 3.1. When built on top of a component-based development model, it is important that all the components can be composed into a functional citizen observatory. The components therefore have to be compatible with one another. The client-side components produce a high variety of data types that need to be compatible with server-side components that process and persistently store the data. These server-side component must then in turn produce output data compatible with the various feedback and visualisations components.

ILR-2: Scalability

Scalability can be measured in various ways, depending on the context. In the case of a CO meta-platform, we particularly focus on two types: load scalability and functional scalability.

Load scalability is the CO meta-platform's ability to easily expand and contract its resource pool to accommodate heavier or lighter loads (e.g., sudden increase or decrease in participants). It also relates to the ease with which a component can be modified, added, or removed to accommodate changing loads.

The meta-platform must be capable of specifying several citizen observatories that each enact multiple campaigns simultaneously. Because a large number of participants will be contributing data to these observatories, appropriate scaling mechanisms must be put in place. For example, if a certain server-side component cannot handle the amount of data it receives from a large number of clients, multiple instances of the overloaded component can be spawned and the input load balanced among the spawned instances. However, special care is required as additional problems are created when dealing with a stateful component, such as one that keeps track of the average value of its input.

Functional scalability is the ability to enhance and extend the CO meta-platform by adding new functionality with minimal effort. Because technologies keep evolving (e.g., smartphones being equipped with additional sensors) it is required to design a citizen observatory construction tool that allows for the addition of new components and the maintenance of existing ones.

The functional scalability of the platform is directly influenced by the components' granularity: fine-grained components provide very basic features and they can be composed into larger, more complex components. Such a composition mechanism facilitates the addition of new features. For example, if smartphones would suddenly be equipped with a new sensor, the CO meta-platform only has to provide a new component that is capable of accessing the sensor and outputting its data. This raw data can then be connected to various already existing data processing components.

3.4 The Right Tool for the Job

Conceptually, a citizen observatory is a distributed architecture where the data generated by participants *flows* from their mobile apps to a server. On this server, incoming data *flows* through a series of data processing components. Intermediate results of the processed data then *flows* back to the mobile apps and web-based dashboard in the form of feedback.

From this point of view on citizen observatories, combined with the aforementioned CO meta-platform requirements (cfr. section 3.3.1 and section 3.3.2), we propose to use a *visual reactive flow-based domain-specific language* (VRFBDSL) that can handle the *distributed* nature of a citizen observatory's architecture. The advantages of using a VRFBDSL are as follows:

- *Flow-based programming* is a flavour of component-oriented programming approach by nature. It enables one to design a citizen observatory as a composi-

tion of reusable and configurable components, resulting in a highly customizable platform (ULR-1). Additionally, a component-based approach increases the functional scalability of the CO meta-platform (ILR-2).

- *Reactive Programming* allows developers to express programs in terms of what to do, and let the language automatically manage when to do it. In the context of the CO meta-platform, this means that the execution of each component used in the implementation of a citizen observatory is triggered automatically based on input data availability, resulting in reactive citizen observatories (ULR-3).
- A *visual programming language* greatly increases the usability of the meta-platform for societal stakeholders and communities with very little programming skills (ULR-2). For example, its visual cues can be used to indicate and ensure compatibility of the various components used to construct the citizen observatory (ILR-1).
- The use of a *domain-specific language* further enhances the usability of the platform (ULR-2), as it ensures that stakeholders are familiar with the domain of the language, rather than forcing them to rely on a general purpose language that is more difficult to understand.

In the remainder of this chapter, we provide a short introduction to each of these programming concepts.

3.5 Flow-Based Programming

Flow-based programming (FBP) was invented by J. Paul Morrison in the early 1970s. The FBP development approach views an application not as a single, sequential, process, which starts at a point in time, and then does one thing at a time until it is finished. Instead, FBP views an application as a network of asynchronous processes communicating by means of streams of structured data chunks [97]. FBP applications are represented as a directed graph consisting of processes as nodes and connections as edges. Processes access connections by means of *ports*. A process is an instance of a component, and runs concurrently with other processes. Multiple instances of the same component can be created. Morrison explains the paradigm as follows [94]: “*An application built using FBP may be thought of as a "data processing factory": a network of independent "machines", communicating by means of conveyor belts, across which travel structured chunks of data, which are modified by successive "machines" until they are output to files or discarded.*”

FBP can be considered a particular style of dataflow programming, but considering the fact that there is a sizeable body of published work related to dataflow programming and/or architectures, the use of the term dataflow may cause some confusion. Therefore, we first introduce the general concept of dataflow programming, and then provide more details on the particular type of dataflow, i.e., flow-based programming, that is used in the context of this dissertation.

3.5.1 Dataflow Programming

Dataflow Programming has been a research topic of programming languages design since the '70s. Several researchers argued that conventional von Neumann processors are inherently unsuitable for the exploitation of parallelism [34, 132]. The main issue with von Neumann hardware is the bottleneck caused by the shared bus between the program memory and data memory [2, 5]. This shared bus prevents the simultaneous access of both program instructions and the program data. As a result, the CPU is forced to wait for data to be transferred from memory, limiting the effective processing speed of the CPU. As a solution, the dataflow architecture [132, 31] was proposed which avoids both of these bottlenecks by using only local memory and by executing instructions as soon as their operands become available.

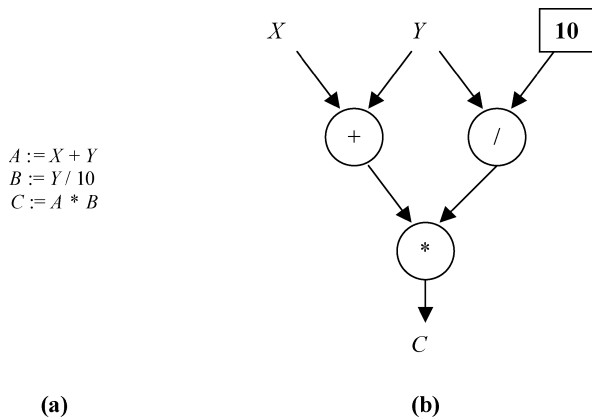


Figure 3.2: A simple dataflow program (a) and its directed graph representation (b) [66].

In the dataflow execution model a program is represented by a directed graph, as illustrated by figure 3.2. The nodes of the graph are primitive instructions such as arithmetic or comparison operations. Directed arcs between the nodes represent the data dependencies between the instructions [75]. Values are propagated to the

dependent nodes as soon as they are processed, triggering the computation of the dependent nodes.

There exists a large body of research devoted to dataflow programming in various application domains; Johnston et al. provide a comprehensive overview of more recent advances in dataflow programming languages [66].

One issue with the term dataflow is that its meaning has become very broad. The term dataflow programming is generally used to refer to *synchronous* dataflow programming, while flow-based programming is used to refer to *asynchronous* dataflow programming.

Synchronous dataflow programming executes a program once by starting with the data of all the inputs, also called sources. Each node in the graph is executed once when the data of *all* of its input ports becomes available. When a node in the graph completes its execution, output data becomes available on all of its output ports. This data then “flows” to the input ports of the downstream nodes allowing them to start executing. This process is repeated until the entire graph (representing the program) is traversed.

Asynchronous dataflow programming resembles the event-based and message passing programming model: the nodes in the graph are connected with asynchronous messaging channels and all of the nodes are continuously waiting for messages to arrive to their inputs. Whereas in synchronous dataflow programming the nodes are executed only once, in flow-based programming the nodes are constantly waiting for new asynchronous messages to arrive from other nodes.

3.5.2 Flow-Based Programming Characteristics and Classification

Flow-based programming is best understood as a coordination language, rather than a programming language [96]. A coordination language embodies a coordination model, which Gelernter and Carriero define as [52] “*the glue that binds separate [computational] activities into an ensemble*”. A coordination model provides operations to create computational activities and communication among them [52]. In the context of flow-based programming, these computational activities are “black box” processes that communicate among predefined connections.

The strength of FBP is that different applications can be built from the same set of components by connecting them in different compositions. In 1974, Nate Edwards coined the term “configurable modularity” [38] to denote the ability to reuse independent components by changing their interconnections. One of the important character-

istics of systems exhibiting configurable modularity is that they can be built using “black box” reusable components [96], as is the case in flow-based programming. FBP is therefore a *component-based development* approach by nature.

Component-based software engineering (CBSE), is a branch of software engineering that emphasises the separation of concerns with respect to the wide-ranging functionality available throughout a given software system [85]. It is a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems [134]. CBSE is primarily concerned with three functions [56]:

- *Developing* software from preproduced parts
- The ability to *reuse* those parts in other applications.
- Easily *maintaining* and *customising* those parts to produce new functions and features.

There are many definitions of what a *component* in CBSE entails. Heineman and Councill [56] define a software component as a “*software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard*”. In other words, a component model defines how to construct an individual component and enforces global behaviour on how a set of components communicate and interact with each other. A component can be composed with another component by creating assembled connections. The general term ‘assembly’ is used, as components can be composed in many different ways. The component model can also define other aspects such as naming conventions, meta data, interoperability, customisation, etc.

Another characteristic of FBP is that its system design is generally split into two layers: the graph layer and the component layer. The graph layer usually has a visual representation and is intended to be used by the graph designer, i.e., it is used to construct the graph by connecting the various components together to form the application’s logic. The component layer contains the actual source code of each component as implemented by the component developer.

Classical vs. Reactive Flow-Based Programming

Flow-based programming has evolved considerably over the last 40 years, and several new FBP systems were recently developed that are missing some key characteristics of true FBP, such as asynchrony, and each process running in its own thread or other concurrency mechanism. As a result, we make a distinction between *Classical FBP* and *Reactive FBP*:

Classical FBP is what we consider to be the original approach. In this case, operating system or virtual machine threads are used, enabling processes to execute concurrently. Connections between processes are usually implemented as bounded buffers or FIFO queues. Processes can choose from which port they want to read data. Classical FBP features blocking reads and writes: a process that reads from an empty input port blocks until input data arrives. Alternatively, a process writes to a connection whose buffer is full, the process is suspended until the buffer is no longer full. Examples include JavaFBP [95] and C#FBP [93].

Reactive FBP is a FBP-inspired approach designed around event-listeners. Processes operate using a publish/subscribe pattern, i.e., processes wait for data arriving on their input ports and publish data on their output ports. Events include both data sends, as well as connects, disconnects, etc. Unlike Classical FBP, processes cannot choose from which port they want to read data. Instead, processes in Reactive FBP simply react to every incoming event on a port in the order of arrival. Examples include NoFlo [8], Node-RED [102], and MicroFlo [91].

3.6 Reactive Programming

Reactive programming (RP) is a programming paradigm that is built around the notion of continuous time-varying values and propagation of change. It facilitates the declarative development of event-driven applications by allowing developers to express programs in terms of what to do, and let the language automatically manage when to do it [7]. In this paradigm, state changes are automatically and efficiently propagated across the network of dependent computations by the underlying execution model. RP introduces the notion of *behaviors* for representing continuous time-varying values and *events* for representing discrete values. In addition, it allows the structure of the dataflow to be dynamic (i.e., the structure of the dataflow can change over time at runtime).

The history of research and developments in reactive programming fall outside the scope of this dissertation. For more details we refer the reader to a comprehensive survey by Bainomugisha et al. [7]. In more recent years, reactive programming has gained a lot of attention in the front-end developers community. As a result, several libraries came into existence that implement RP principles. Examples of such libraries include Bacon.js [6], Meteor [61], React.js [60], and RxJS [125].

3.7 Visual Programming Languages

Visual programming languages (VPL) are languages that use some form of visual representation instead of — or in addition to — textual representations used by traditional programming languages [115]. One well-known example is Scratch [110], which represents programming statements graphically as colourful pieces of a jigsaw puzzle that can be composed to define the logic of a program. An example of a Scratch script is illustrated in fig. 3.3.

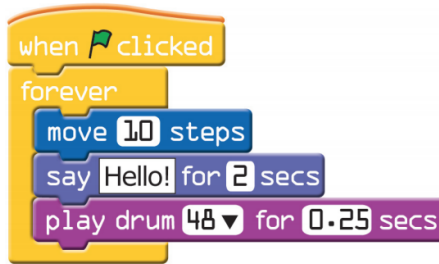


Figure 3.3: Sample Scratch script [110].

Visual programming languages have some specific goals, which most commonly are [14]:

- Making programming more understandable to non-technical users.
- Improving the correctness with which users perform programming tasks.
- Improving the speed with which users perform programming tasks.

It is important to distinguish visual programming from visual aids for programming. The latter is a more general term which also includes tools for program code visualisation (e.g., class hierarchies, state diagrams, etc.) and data visualisation, as illustrated by the taxonomy of visual aids for programming depicted in fig. 3.4. Visual programming systems allow the programmer to construct programs using techniques that display computational functions and elements in two or more dimensions. Program code visualisation, on the other hand, helps in debugging and understanding programs by providing visualisations of various aspects of the program. Examples of program visualisation include the Unified Modeling Language (UML) or Architecture Description Languages.

Visual programming can be further subdivided into two key branches [119]: graphical interaction systems and visual language systems. This division is based upon how the graphics are used to build the program. In graphical-interaction

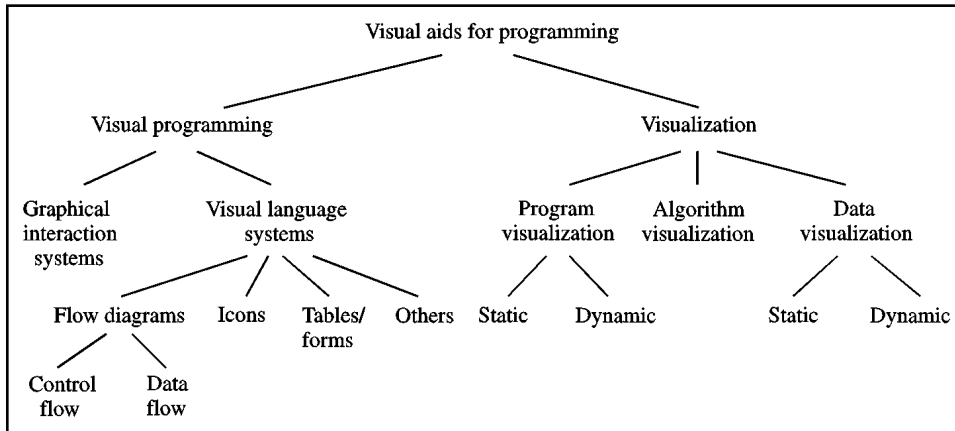


Figure 3.4: Taxonomy of visual aids for programming [119].

systems the user guides or instructs the system in order to create the program. Visual-language systems use icons, symbols, charts, or forms to specify the program. The main difference between these two systems is that in the former the user interaction with the system is important, whereas in the latter the arrangement of symbols on the screen is important. Visual-language systems are further classified depending on the graphical abstraction used. In this dissertation, we are mainly interested in data flow diagrams, i.e., visual data flow programming languages.

3.7.1 Visual Data Flow Programming Languages

Data-Driven Nets [29], developed in the 1970s, is a graphical programming concept that can arguably be considered as the first visual dataflow programming language (VDFPL). However, it operated on such a low level that it was not practical to program directly using this language. As a result, a higher-level language was developed known as Graphical Programming Language [30]. In this language a program is depicted as a graph, and every node in the graph is either an atomic node or a compound node that can be expanded to reveal a sub-graph. These subgraphs can be defined recursively. Arcs in the graph are typed and the whole environment has facilities for debugging, visualisation, and text-based programming [66].

Most visual data flow programming languages use boxes to represent functions and lines to represent the flow of data between functions. This representation maps

directly onto the flow-based programming concepts of processes and connections. It is therefore not surprising that many FBP frameworks have a visual representation.

3.8 Domain-Specific Languages

General-purpose languages such as C, Java, and Python enable programmers to write instructions for computers to solve problems in many domains. The broad applicability of these languages results in a lack of specialised features for a particular domain. Hence, when the problem is limited to a certain domain, a more specialised language is required that provides features created specifically for that domain, i.e., a *domain-specific language*.

A Domain-Specific Language (DSL) is a language that is optimised for a given class of problems called a domain [129]. A DSL is based on abstractions that are closely aligned with the domain for which the language is built. As such, a DSL includes a syntax suitable for expressing these abstractions concisely. This syntax can be either textual or graphical. Sometimes, representations such as graphical diagrams, matrices, and tables are used alongside text to provide the desired closer mapping to the problem domain. There is a wide variety of DSLs, ranging from widely used languages for common domains, such as HTML for web pages or SQL for relational database queries, down to languages used by only one or a few pieces of software, such as MATLAB. DSLs have to sacrifice some of the flexibility of general-purpose languages (GPLs) to ensure productivity and conciseness of relevant programs in a particular domain. They can also be restricted on purpose to only allow the creation of correct programs. Unlike GPLs, there is no requirement for DSLs to be Turing complete [122].

Domain-specific languages can be used for a variety of purposes, such as being a utility for developers to automate a specific aspect of software development or, in larger-scale systems, to describe the architecture of a software system. Voelter et al. [129] provide an in-depth book on the design, implementation and use of DSLs. In the context of this dissertation, we are mainly interested in the use of DSLs for *Domain-Specific Modelling*.

3.8.1 Domain-Specific Modelling

Domain-Specific Modelling is a software engineering methodology that involves the systematic use of a (graphical) DSL to design and develop systems. Modelling in this context refers to *prescriptive* modelling, where a model is created that can be used to

(automatically) construct the target system. This is in contrast to a descriptive model that represents an existing system for discussion and analysis.

Domain-Specific Modeling mainly aims to accomplish two goals [72]: first, raise the level of abstraction by specifying the solution in a language that directly uses concepts and rules from a specific problem domain. Second, generate final products in a chosen programming language or other form from these high-level specifications (i.e., automatic creation of executable source code from the domain-specific model).

There are many benefits of using DSM, such as increasing the productivity and quality of the created product through the removal of (unnecessary) degrees of freedom for programmers, the avoidance of duplication of code, and the consistent automation of repetitive work by the execution engine. Validation and verification of the created product are easier to implement, as the error messages can be more meaningful through the use of domain concepts. Using DSM, developers only need to model the core logic of an application system independent of the underlying technology platform, making it possible to change the execution engine of the model to execute the code on a new platform. But perhaps the most interesting benefit of DSM is *domain expert involvement*: if the abstractions are closely aligned with how domain experts express themselves, the domain experts, who often are non-programmers, can become developers and use the DSL to write the program code by making complete specifications using familiar domain concepts. This capability to support domain experts' concepts makes DSM very applicable for end-user programming.

3.9 Related Work

Although the concept of a citizen observatory meta-platform is, to the best of our knowledge, unprecedented, our approach is rooted in the literature of visual data flow programming languages.

Node-RED [102] is a reactive flow-based programming tool built on top of Node.js. It provides a browser-based drag-and-drop editor — depicted in fig. 3.5 — for wiring together hardware devices, APIs, and online services. Node-RED provides a library where people can add new components and share existing component compositions.

Other examples of reactive FBP include NoFlo [8], MicroFlo [91], and MsgFlo [98]. Each of these FBP implementations provide a visual editor, although they adopt a slightly different approach compared to Node-RED: rather than having their own dedicated visual data flow programming language, they both rely on Flowhub [46], a technology agnostic (e.g., supports any runtimes compatible with

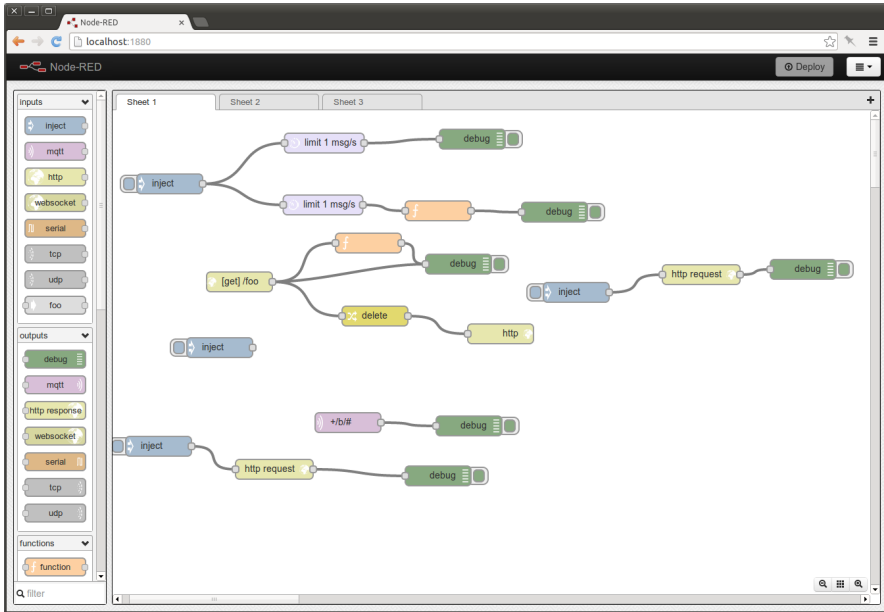


Figure 3.5: Node-RED's browser-based editor.

the FBP protocol) web-based IDE for flow-based programming. The web-based IDE of FlowHub is depicted in fig. 3.6.

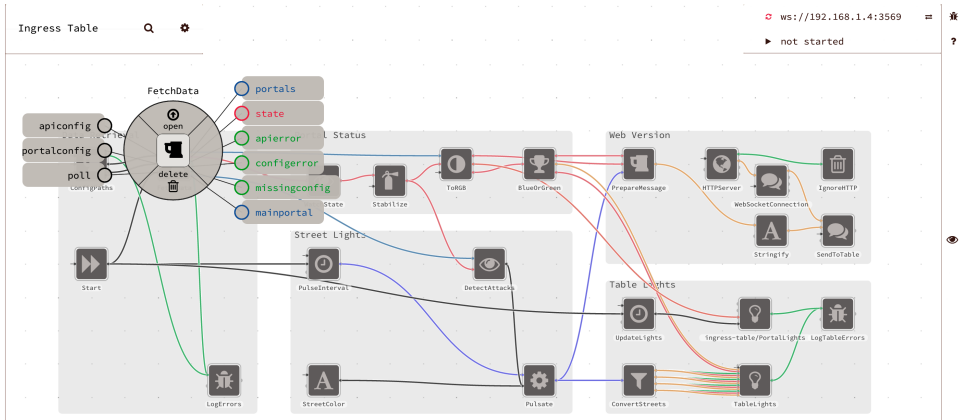


Figure 3.6: FlowHub IDE.

Despite their identical visual editor, each of these FBP implementations have a different domain. NoFlo is based on Node.js and is written in JavaScript and CoffeeScript. It can be used to build server-side or client-side (browser) applications.

MicroFlo is written in C++ and is intended to be used on microcontrollers and embedded devices. MsgFlo components can be implemented in any language (to reuse existing code or libraries) and can be used to implement a distributed FBP system spanning multiple computers/devices.

VDFPLs have seen adoption outside of the domain of FBP as well. Perhaps one of the best known examples of a VDFPL is LabVIEW [62]. First released in 1986, it allows the construction of programs for data analysis in laboratories. LabVIEW programs are called virtual instruments, because their appearance and operation often imitate physical instruments. Creating a new virtual instrument is done by specifying the front panel (the user interface of the virtual instrument), and the block diagram (containing the graphical source code). An example block diagram is depicted in figure 3.7. Constructing the block diagram is done by connecting different function-nodes, displayed as boxes with icons, using wires representing the data paths. LabVIEW uses “G”, a dataflow programming language, meaning that the wires propagate variables and any node can execute as soon as all its input data become available.

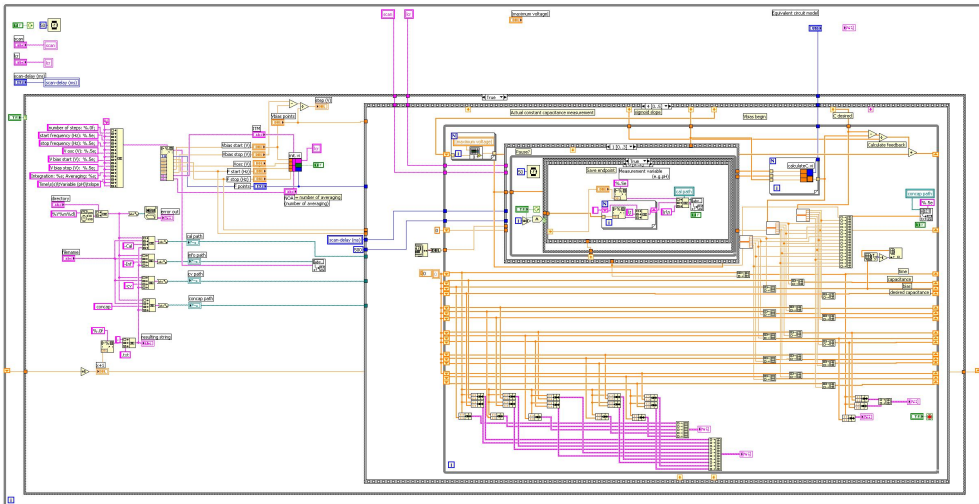


Figure 3.7: Example of LabVIEW Block diagram.

Another example is Reaktor [63], a graphical modular software music studio that lets musicians and sound specialists design and build their own instruments, samplers, effects and sound design tools by connecting various modules. The Reaktor environment is depicted in figure 3.8.

Blender [47] is an open-source 3D computer graphics toolset for creating video games, visual effects, animated films, etc. Blender allows you the creation of a mate-

Chapter 3: Towards a Citizen Observatory Meta-Platform: Requirements

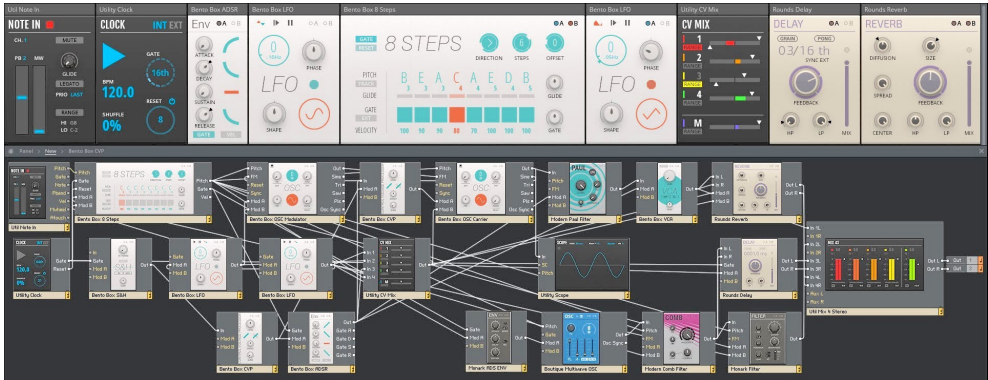


Figure 3.8: The Reaktor environment.

rial (i.e., the artistic qualities of the substance that a 3D object is made of) by routing basic materials through a set of nodes. Each node performs some operation on the material, changing how it will appear when applied to the mesh, and passes it on to the next node. Figure 3.9 depicts this node editor environment.

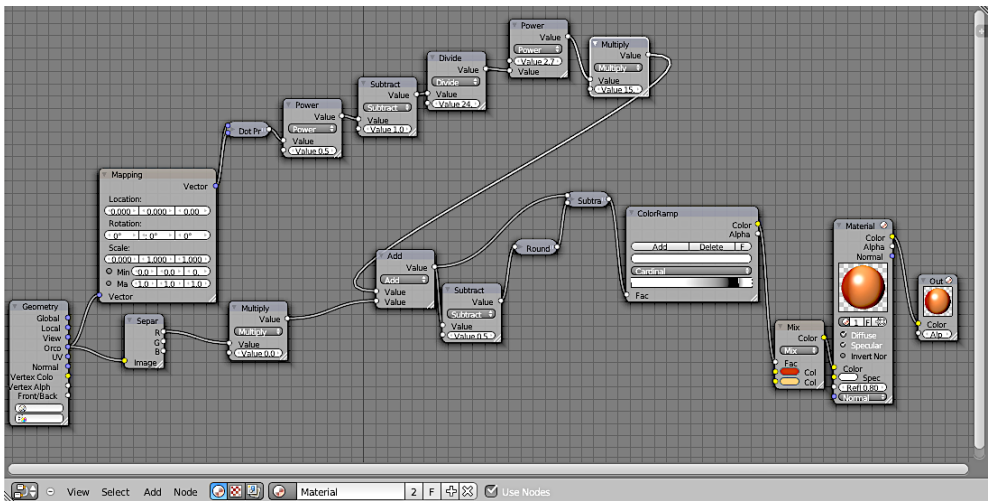


Figure 3.9: The Blender Node Editor.

These, and other examples [131, 135], make it obvious that VDFPLs are applicable in many domains. However, many of these systems did not focus directly on research in dataflow programming. Instead, they were produced to solve specific problems and use the dataflow model because it provides the best and/or most intuitive solution to a specific problem.

3.10 Conclusion

Designing a citizen observatory meta-platform that enables domain experts to create their own citizen observatory is a challenging task. This becomes apparent by analysing various requirements of the meta-platform, i.e., customisability, usability, reactivity, compatibility, and scalability. Using these requirements as a motivation for our decision, we advocate the use of a visual flow-based domain-specific language that enables non-ICT stakeholders to build citizen observatories. This chapter introduced the programming concepts that facilitate the implementation of the CO meta-platform, i.e., flow-based programming, reactive programming, visual programming languages, and domain-specific languages.

Flow-based programming is particularly suited to ensure customisability of a citizen observatory by providing the end-user with a toolkit of relevant participatory sensing components that can be composed to provide the desired features. FBP also aids in the functional scalability of the meta-platform as newly designed components can easily be integrated in the rest of the system. Additionally, FBP ensures the reactivity of each citizen observatory, enabling real-time data processing and participant coordination.

By incorporating reactive programming principles, the execution of components is triggered automatically based on input data availability, resulting in highly reactive citizen observatories that can provide participants with immediate feedback.

Usability of the CO meta-platform can be facilitated through a visual programming language, enabling end-user programming by ICT-agnostic stakeholders. Visual cues can prevent incompatible component compositions.

However, simply providing a visual environment for the domain-expert is not sufficient. The components presented in the visual programming language have to make sense to the domain experts. Therefore, a (visual) domain-specific language that only presents relevant concepts to the end users is needed.

Although this chapter discussed various existing visual data flow programming languages, none of these are suitable to implement a citizen observatory meta-platform for the following reasons: first, none of these languages enable the design of a citizen observatory as the necessary domain-specific components to implement PS data gathering and processing are not present. Second, the existing languages either provide no support for distribution, or those that do only provide low-level components that are too difficult to configure for non-ICT experts. This makes the implementation of citizen observatories more difficult due to their distributed architecture (i.e., mobile apps, processing server, web based visualisations).

Chapter 3: Towards a Citizen Observatory Meta-Platform: Requirements

In the next chapter, we introduce a novel visual reactive flow-based domain-specific language, named DISCOPAR, which is designed specifically to cope with these issues. This language is used throughout our citizen observatory meta-platform.

4

LANGUAGE CONCEPTS OF DISCOPAR

One of the main challenges of a citizen observatory meta-platform is ensuring usability by (ICT-agnostic) societal stakeholders and communities (cfr. section 3.3.1 - ULR-2). Deploying a citizen observatory and setting up campaigns should therefore be possibly without or with limited programming skills. To make this possible, we propose DISCOPAR (Distributed Components for Participatory Campaigning), a new visual flow-based domain-specific programming language created specifically to hide the non-essential complexity of citizen observatories and their distributed nature from the end-user, and to present only concepts that are relevant to their domain. DISCOPAR is used throughout the meta-platform to construct every part of a citizen observatory, i.e., the mobile data gathering app, server-side data processing, and web-based visualisations can all be set up using a single visual language. Due to its flow-based nature, DISCOPAR's design is split in two layers: a graph layer enabling end-users to visually construct observatories, and a component layer containing each component's source code and the necessary abstractions to compose them.

This chapter focusses on DISCOPAR's graph layer. More specifically, it presents the visual programming language that is built on top of DISCOPAR's component layer. Details on the more technical component layer are discussed in chapter 6. In this chapter, we first describe the general idea of DISCOPAR and introduce DISCOPAR^{DE}, our web-based visual programming environment for designing programs in DISCOPAR. Next, we introduce the various concepts and visual syntax of DISCOPAR. Then, we provide more details on DISCOPAR^{DE}. We end the chapter by introducing four common strategies of visual programming languages that facilitate

end-user programming, and describe how these strategies are applied to DISCOPAR and DISCOPAR^{DE}.

4.1 Programming with DISCOPAR

In section 3.4, we proposed to implement our CO meta-platform using a visual reactive flow-based domain-specific language (VRFLDSL) that can handle the distributed nature of a citizen observatory's architecture. The idea to use a VRFLDSL was motivated through the analysis of both user level requirements (cfr. section 3.3.1) and implementation layer requirements (cfr. section 3.3.2) of the CO meta-platform. These requirements, combined with the distributed architecture of citizen observatories, highlight the significant challenge involved in creating a CO meta-platform capable of constructing any type of citizen observatory.

When analysing existing flow-based languages, we noticed that most FBP languages do not allow the flow to be partitioned across multiple devices. Those that do are either research prototypes [10] or only provide low-level communication components that have to be manually configured by the end user [98]. Configuring these components involves details such as specifying the IP-address and port to establish a connection. Such details are too difficult to understand for ICT-agnostic users, who are incapable of programming such technicalities. What is needed is a distributed flow-based language where stakeholders only need to express themselves with language concepts that come from the domain of citizen observatories and participatory campaigning. As a result, we created DISCOPAR, a new visual reactive flow-based domain-specific programming language with support for distribution.

DISCOPAR ensures usability by ICT-agnostic stakeholders through the use of domain-specific components, which can be composed in a visual manner. Through such component composition, stakeholders are capable of implementing the various elements of a citizen observatory without having to worry about the underlying complexity of the CO's distributed architecture. Due to the lack of support for distribution in existing FBP languages, we opted to design a language from the ground up. By creating a new language instead of implementing additional abstractions in an existing FBP language, we facilitate the understanding and development of mechanisms that can handle the distributed architecture of a citizen observatory. To the best of our knowledge, DISCOPAR is the only distributed visual flow-based language where distributed connections are automatically handled by the system and do not require any configuration from the programmer.

Programs in DISCOPAR are a *directed acyclic graph* (DAG) where each node of the graph consist of a component instance and where the edges are implemented as real-time data streams. Programs in DISCOPAR can be visually assembled through our web-based visual programming environment, named DISCOPAR^{DE}. Figure 4.1 depicts an instance of DISCOPAR^{DE}, consisting of a component menu from which graph designers can select components and visually assemble them on the canvas through drag-and-drop interactions. More details on the features of DISCOPAR^{DE} are deferred to section 4.3.

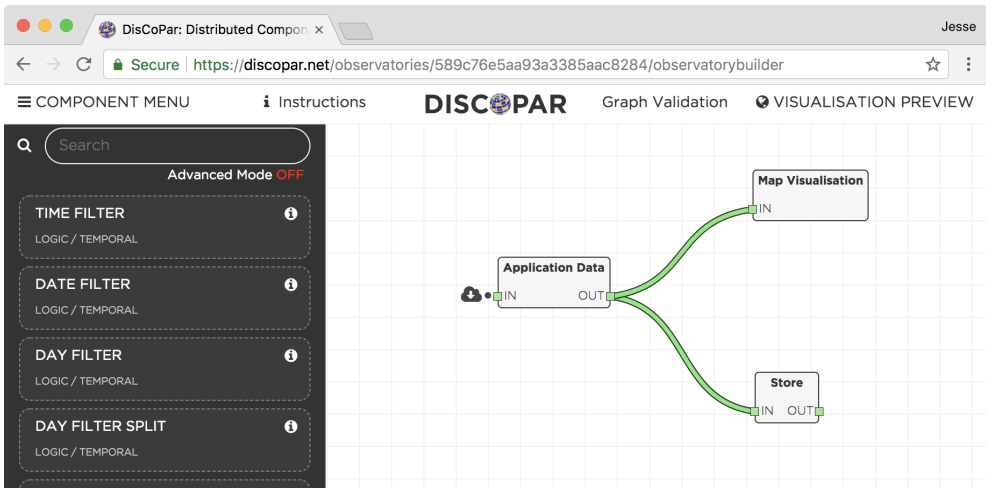


Figure 4.1: DISCOPAR^{DE}: DISCOPAR’s web-based visual programming environment.

Due to its flow-based nature, DISCOPAR’s design is split in two layers: a graph layer enabling end-users to visually construct observatories, and a component layer containing each component’s source code and the necessary abstractions to compose them. Figure 4.2 illustrates the interaction between DISCOPAR’s graph layer and component layer. DISCOPAR^{DE}, DISCOPAR’s integrated web-based visual programming environment, enables ICT-agnostic users to program an application using the visual syntax of DISCOPAR’s graph layer. To execute a program designed in DISCOPAR^{DE}, the constructed DAG is passed onto the execution engine of DISCOPAR, named DISCOPAR^{EE}. DISCOPAR^{EE} is responsible for executing the DAG. This is done by initialising the components by loading their source code from the component layer, and establishing the real-time data streams between the newly created processes. Due to its reactive flow-based nature, each process in the DISCOPAR^{EE} activates automatically based on the availability of data on its incoming data streams.

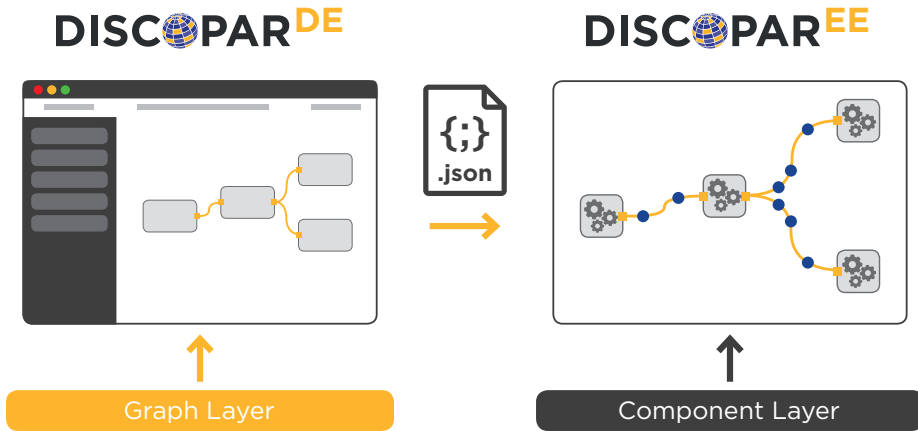


Figure 4.2: Overview of DISCOPAR.

Each element of a citizen observatory’s distributed architecture is implemented through DISCOPAR. This means that the CO’s mobile app, server-side data processing, and web-based visualisations all execute a different DAG in DISCOPAR^{EE}. More details on how the CO meta-platform enables each element of a citizen observatory to be programmed through DISCOPAR^{DE} are explained in chapter 5.

It is important to realise that DISCOPAR supports distribution and hides any technological complexity associated with it. DISCOPAR^{DE} is capable of simultaneously programming client-side and server-side logic, and graph designers can create a connection between those in the same DAG in the same way that they create a connection between those in the same DAG, i.e., by performing a drag-and-drop interaction. Connections made between components residing on a different device are automatically handled by DISCOPAR. In other words, a DAG deployed on one particular device (e.g., mobile app) transparently establishes a connection with a DAG on another device (eg server-side data processing). More details on how DISCOPAR automatically handles these distributed connections are provided in chapter 6.

4.2 Concepts and Terminology

This section provides a general introduction to various flow-based programming concepts and introduces their visual syntax in the graph layer of DISCOPAR.

4.2.1 Components

Components form the basic building blocks used by developers to build an application using flow-based programming. In FBP, the emphasis in implementing an application shifts from building everything from scratch to connecting pre-existing components and only creating new ones when absolutely required. Implementing components is commonly done using classes, functions or small programs in conventional programming languages. Unlike components in dataflow programs, which primarily consist of primitive arithmetic and comparative operations (cfr. fig. 3.2), components in FBP are more complex. FBP uses “black-box” components, referring to the fact that the application developer does not need to understand or modify the internal implementation of a component. Programming in FBP is done by composing various components into a DAG to form the desired logic.



Figure 4.3: Example of DISCOPAR’s visual syntax for components.

DISCOPAR features a variety of components that all share the same visual syntax. As an example, the visual representation of some components is illustrated in fig. 4.3. Components are represented as boxes labelled with their name. Components can have multiple input and output ports, which are represented as little squares on the side of a component. Input ports are always shown on the left side of a component, while output ports are shown on the right.

4.2.2 Processes

An instance of a component is called a *process*. A process is an asynchronously executing piece of logic. Processes can be handled by the system using threads or a similar form of concurrency, or at least provide the illusion of it to the designer. Multiple processes of the same component can be simultaneously active. A process is stateful and can access its own internal state and ports, but it cannot access other processes. Processes communicate by sending and receiving structured data chunks called *information packets* (IP). A process is activated when it receives an informational packet on one of its input ports. Exceptions to this rule are processes that are automatically activated on application startup.

Processes in DISCOPAR are spawned behind the scenes, so technically speaking they do not have a visual syntax. The graph designer implements an application by creating a visual representation of interconnected components in DISCOPAR^{DE}. The DAG of interconnected components is then loaded by DISCOPAR^{EE}, where processes are initialised that communicate across the predefined connections.

4.2.3 Connections

Processes communicate by means of *connections*. Connections can be considered streams on which information packets “flow” from one process’s output port to another’s input port. In FBP, connections can be implemented using bounded buffers or FIFO queues. The size of this buffer or queue is known as connection capacity, which some FBP implementations allow to be 0, meaning that the IPs are transferred immediately between the sending and receiving processes.

The visual representation of connections in DISCOPAR is illustrated in fig. 4.4. They are represented as lines connecting an output port to an input port. The colour of a connection is always the same as the colour of the output port that the connection originates from. An output port’s colour indicates the type of data the port emits (cfr. section 4.2.4).

Some ports in DISCOPAR only accept distributed connections that are automatically established by the system. For example, the output port of the Upload component is automatically connected to the observatory’s server-side whenever an internet connection is available. This is indicated to the graph designer by a cloud icon (cfr. fig. 4.4).

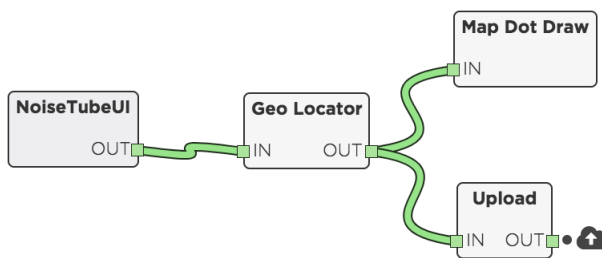


Figure 4.4: Visual syntax of connections in DISCOPAR.

A component’s input port can have multiple incoming connections. However, the number of connections attached to an input port of a component does not impact the behaviour of that component. Once the component is initialised by DISCOPAR^{EE},

the corresponding process will *react* to any IP, regardless of its origin, as soon as it arrives on the input port.

In classical FBP (cfr. section 3.5.2), an information packet is tracked from its creation to destruction, and can only be “owned” by a single process at a time or be in transit between processes. This means that in classical FBP a single output port can only be connected to a single input port, as otherwise the same IP would be owned by multiple processes at the same time. Classical FBP requires IPs to be explicitly copied using a dedicated component with multiple output ports that are then connected to multiple input ports. In order to not needlessly complicate matters for the graph designer, DISCOPAR adopts the reactive FBP approach where a single output port can be connected to multiple input ports, thereby implying automatic replication of data.

4.2.4 Ports

The contact points between processes and connections are called *ports*. Every port is named to enable FBP components to refer to them without needing to be aware of what they are connected to. A component can have multiple inputs or outputs. A process can send to or receive from any of its ports. Input ports provide *receive* functionality to dequeue IPs from a connection’s buffer. Output ports provide *send* functionality to queue an IP into the port of a connected process.

A process reacts to the arrival of data on an input port by executing code from its black box implementation. This code can potentially produce output, which is then published on one of its output ports. The behaviour of components with multiple input ports depends on the implementation of the component. More details on this are provided in section 6.1.3.

It is possible for a component to not have an input port, in which case it acts as a *source* of the DAG. A source is a component that automatically produces output. For example, a component generating output from a smartphone sensor. Similarly, a component without an output port acts as a *sink* of the DAG. A sink is a component that consumes data, such as a linechart component that draws the consumed value on a chart.



Figure 4.5: DISCOPAR’s supported data types and corresponding colours.

Ports in DISCOPAR are typed. Ports can only be connected if they “understand” each other. Components are compatible if the output of one component can serve as input of another component. For example, it makes no sense to send a numerical value to a component expecting JSON input. The colour of the port indicates what type of data the component can receive. The various types that are currently supported, along with their corresponding colour, are shown in fig. 4.5. The `Any` type is a special case and accepts connections from any output port, regardless of its type.

4.2.5 Information Packets

An information packet (IP) is data that is sent from one process to another. An IP has some affinities with the concept of “object” in Object-Oriented Programming. Information packets have a life cycle and are owned explicitly by one process at a time. The actual implementation and content of an IP depends on the application domain. One special type of information packet, called *initial information packets*, are not meant to be passed around between processes. They only contain configuration information and are given to a particular process when the program is started by the DISCOPAR^{EE}. For example, a component capable of establishing a serial connection to external devices through USB requires that the USB protocol details are passed to the process upon initialisation.

In DISCOPAR, IPs can be numbers, strings, or JSON objects. However, these are often not presented to the user as-is, but use an additional layer of abstraction so that users are able to deal with domain specific data instead (cfr. port typing in section 4.2.4).

Citizen observatories can gather data on a variety of topics, such as noise pollution or eating habits. Despite this diversity, there are some commonalities between the data collected in every citizen observatory. Observational data typically contains the following information [118]:

- Geographical information, indicating *where* the data sample was made.
- Observation time, indicating *when* the data sample was made.
- Observer, indicating *who* made the data sample.
- Observation Procedure, indicating *how* the data sample was made.
- Observed Data, the measured data of this sample.

DISCOPAR supports this observational data model, and provides a special type of IP that is referred to as an *observation*. Observations can be considered the most

important type of information packet in a citizen observatory. Observations are always created by data-producing processes on the mobile app, such as those reading out sensor values or outputting user input from questionnaires. Observations are augmented with relevant meta-information, such as user id, device model, so that they can be processed and persistently stored in the citizen observatory. Observations can be considered a summary of the captured data. Many sensor components also output the individual values captured in the observation on different output ports. One such example is the `SensorDrone` component, illustrated in fig. 4.6. This component collects sensor readings on temperature, humidity, and atmospheric pressure on a regular interval from an external device that can be connected to the smartphone using Bluetooth. On each interval, these sensor readings are wrapped inside an observation, augmented with meta-data, and sent to the `out` output port. At the same time, each individual numerical value is sent to the corresponding output port.

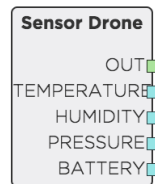


Figure 4.6: The `SensorDrone` component.

4.2.6 Graphs

The structure of an application in flow-based programming is represented as a directed graph. The nodes of the graph are processes, while the edges are the connections between ports. Unlike other FBP languages that do allow the application's directed graph to contain cycles, DISCOPAR adopts a more strict policy where a program is implemented as a directed acyclic graph. This choice was made deliberately to prevent ICT-agnostic users from creating incorrect programs.

Constructing a graph in FBP can either be done graphically using a visual tool, or using a textual Domain-Specific Language. In DISCOPAR, graphs are created through DISCOPAR^{DE}, our web-based visual programming environment.

A FBP application represented through a graph can be executed in the underlying FBP language or library by initialising the processes and establishing the appropriate connections. In the case of DISCOPAR, this is done by loading the graph constructed with DISCOPAR^{DE} in DISCOPAR^{EE}.

Graphs in DISCOPAR do not necessarily deploy every component on the same device (hence the term "distributed" in the DISCOPAR acronym). However, this distribution of components is hidden from the developer. For example, consider the graph depicted in fig. 4.7. This graph filters the observations uploaded by mobile devices based on whether those observations were made within the boundary of a certain city. Observations satisfying this constraint are then aggregated into a map, and this map is visualised to end-users.

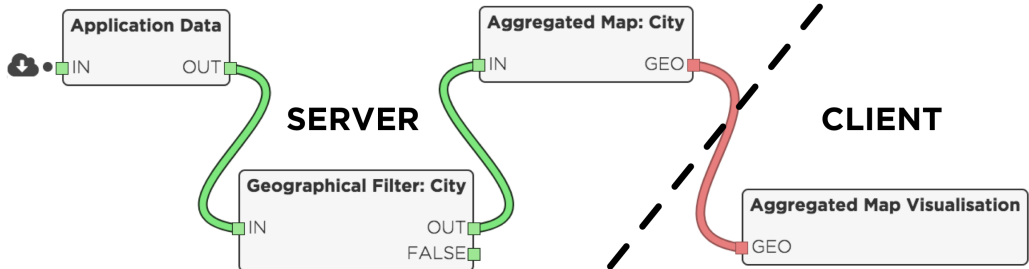


Figure 4.7: A graph containing distributed components.

The graph designer that implemented this data processing and visualisation logic did so only by interconnecting these four components. The data filtering and aggregation processes reside on the server, as they are CPU-intensive operations that process the uploaded data of *every* participant. However, visualisation components in DISCOPAR provides real-time and interactive visualisations that are rendered on the client. Despite this distributed setting, the graph designer can simply drag connections from one (server-side) component to a (client-side) component. Behind the scenes, DISCOPAR will automatically establish the distributed connection between the client and server. More details on how these distributed connections are handled are discussed in section 6.1.6 .

4.3 DISCOPAR^{DE}

In addition to the visual syntax for components, ports, and connections, DISCOPAR features a web-based *visual programming environment*, named DISCOPAR^{DE}, that provides users with the means to visually compose components into a graph through drag-and-drop actions. Although already briefly introduced in section 4.1, this section provides more details on the features of DISCOPAR^{DE}.

The user interface of DISCOPAR^{DE}, depicted in fig. 4.8, consists of four main elements: the component menu, the canvas, a pop-up window granting access to pro-

cess configuration, and the graph validation indicator. This default interface of the DISCOPAR^{DE} can be modified and extended for each particular use throughout the CO meta-platform, as we will discuss in chapter 5.

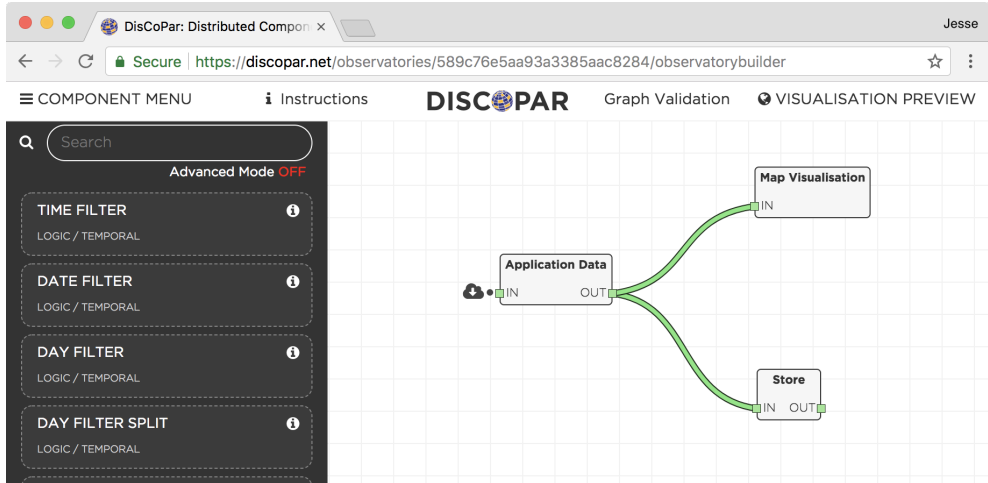


Figure 4.8: DISCOPAR’s default visual programming environment.

4.3.1 Component Menu

The component menu is presented as a searchable list containing the various available components. Searching the available components can be done by name, (sub)category, or on any keyword in a component’s description. This description — toggleable by an information button next to each component — offers more details to the graph designer on the features that a certain component provides.

The set of available components is configured upon initialisation of DISCOPAR^{DE}, meaning that every component is not always available. For example, components that only work on the client-side, such as those producing output from a smartphone sensor, are not made available when using DISCOPAR^{DE} to program the server-side of a citizen observatory. An exhaustive list of every component available in the current implementation of DISCOPAR is available in appendix A.

Components that provide more low-level features are hidden from the component library’s list by default. These components generally are intended to be used by more advanced users. They become available after switching on the advanced mode, which can be done through a button beneath the search bar.

Clicking on a component in the library will create an instance of that component, which immediately appears on the designer canvas. Multiple instances of the same component can be created.

Dynamic Menu Population

An interesting feature of DISCOPAR^{DE} is that the component menu can be dynamically adjusted based on other components in the graph. Graphs constructed in DISCOPAR^{DE} are analysed to extract meta-information, i.e., information about the graph itself, such as the types of data that is produced by a mobile app. This meta-information is used as initial information packet (cfr. section 4.2.5) to configure processes. For example, the `ObservationData` component extracts all the data from an observation, and outputs the individual values on corresponding output ports. This component needs the meta-information to know how many and what type of output ports it should create, which is based on the data produced by the mobile app.

Furthermore, this meta-information can result in additional components being added to the menu. For example, if by analysing a mobile app the system realises that it consists of a survey containing various questions, the system will automatically add extra components to the library which can filter observations based on the input of participants. If a survey asks for the participant's gender, then a component is added to the component menu that can sort observations by gender.

4.3.2 Canvas

The canvas contains the visual representations of the various components, and provides the means to connect their various input and output ports through drag-and-drop interactions. When right-clicking a component on the canvas, a menu appears, shown in fig. 4.9, that enables the graph designer to delete the component or open its configuration window (cfr. section 4.3.3).

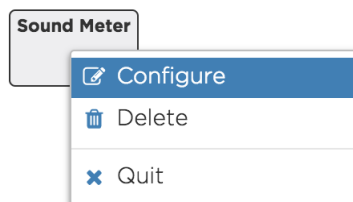


Figure 4.9: Menu shown when right-clicking a component on the canvas.

Every action performed by the designer is immediately and persistently updated in the DISCOPAR graph. There is no need to “save” a graph, and no risk for the designer to lose any modifications to a potentially large graph.

Port Typing

Remember from section 4.2.4 that ports in DISCOPAR are typed, and that ports can only be connected if the output of one component can serve as input of another component. DISCOPAR^{DE}'s canvas includes a visual feedback mechanism that helps the graph designer by presenting this port typing constraint on connections in a visual way: when dragging a connection from an output port, only input ports with matching colours are highlighted and actually accept the connection. The exception to this rule are input ports of the Any type, which are always highlighted as they accept any type of input. DISCOPAR^{DE} thus prevents the creation of a graph in which two components are connected through incompatible ports.

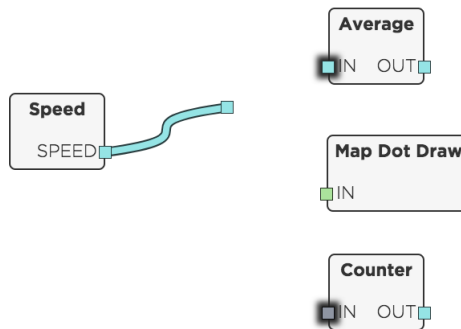


Figure 4.10: Highlighting of compatible ports.

Figure 4.10 demonstrates this colour matching principle. In this example, the designer is creating a connection from the OUT port of the Speed component. This port outputs numerical data, namely the speed a smartphone is moving at based on GPS data. Here, there are two compatible ports: the IN port of the Average component that accepts numerical data, and IN port of the Counter component that accepts any type of data. Thus, both these ports are automatically highlighted when dragging a connection from the Speed component’s output port. The MapDotDraw input port requires observations as input, and can therefore not be connected to the Speed component.

4.3.3 Process Configuration

Components can include configurable settings that enable the designer to customise the behaviour of the process once the component is initialised. When available, these settings are automatically added to the configuration window of each component. Figure 4.11 depicts the configuration window of a `LineChartVisualisation` component. This component visualises numerical values that it receives, and can be configured to change the number of values that are shown simultaneously, the scale of the Y-axis, etc.

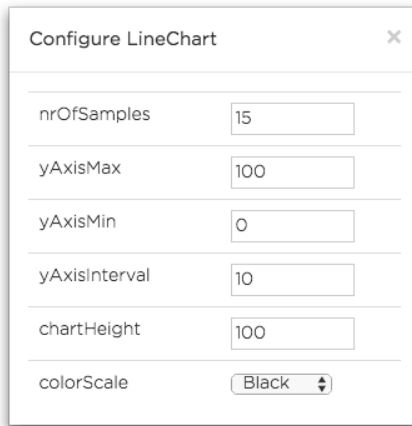


Figure 4.11: LineChartVisualisation component configuration window.

Each component drawn on the canvas in the DISCOPAR^{DE} corresponds to one process of that particular component in the DISCOPAR^{EE} once the graph is deployed. Each process has its own configuration, so it is possible for two processes of the same component to behave differently by using other settings. For example, the graph designer can add two `LineChartVisualisation` components on the canvas but configure them differently. Since we are actually configuring settings that will have an impact on one particular process, we refer to this mechanism as process configuration rather than component configuration.

Components can modify the default configuration window to include a more advanced editor for process customisation. One such example is the `Form` component, which enables the creation of questionnaires that enable data gathering through user input. Figure 4.12 shows the modified configuration window used by the `Form` component to design questionnaires. This form builder consists of a drag-and-drop interface, along with a collection of form elements that can be dragged onto a live preview

The image shows a 'Configure Form' window with a close button (X) in the top right corner. Inside the window, there is a section titled 'Add Field:' with a list of field types: Single Line Text, Multi Line Text, Select Box (Drop down list), Radio Buttons, Checkboxes, Photo, and Rating. Below this list, a 'Text' field is selected and configured. The configuration includes a 'Label' field with the text 'What is your name?', a 'Name' field with the text 'name', and a 'Required?' checkbox which is checked. There is a 'Delete Field' button to the right of the 'Text' field and a 'Save Form' button at the bottom left of the configuration area.

Figure 4.12: Customised component configuration window.

of the form. Upon completion, the form elements are serialised and stored inside the corresponding `Form` instance. The component is then added to the canvas where it can be assembled into the mobile app.

4.3.4 Graph Validation Indicator

DISCOPAR^{DE} supports the concept of graph validation. A graph is considered valid if it satisfies every constraint that is imposed on the graph. These constraints are specified behind the scenes, as we will describe in more detail in section 6.1.5. One example is a constraint for the mobile app logic stating that there must be at least one data connection arriving at the `Upload` component, as otherwise no data will be contributed to the observatory. Another example is testing if a path exists between the sensor components (i.e., the graph's sources) and the destination.

Whenever a constraint is not satisfied, a visual cue is shown to the graph designer to indicate that there are still some unresolved issues, as illustrated in fig. 4.13. Whenever a change is made to the graph, the constraints are re-evaluated and the visual cues updated when necessary.

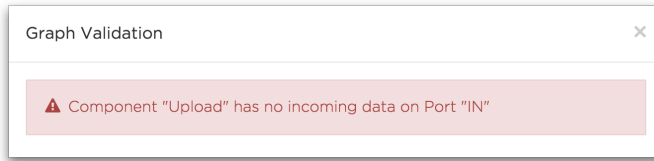


Figure 4.13: Graph Validation error message example.

4.3.5 DISCOPAR^{DE} as Live Programming Environment

The previous section introduces the various features of DISCOPAR^{DE}, the visual programming environment that creates DAGs that are initialised and executed by the DISCOPAR^{EE}. A previously unmentioned and optional feature of DISCOPAR^{DE} is *live programming*. Live programming environments, such as Smalltalk [54], allow changes to the code of a running application in order to provide earlier feedback to the programmer for both debugging and development.

DISCOPAR^{DE} has the option to be configured in such a way that any changes made to the DAG, such as adding a component or changing configuration settings, are immediately, and automatically, mirrored by the DISCOPAR^{EE} responsible for executing that DAG. The DISCOPAR^{EE} mirroring these changes can either be deployed on the same device as the DISCOPAR^{DE}, or on a different device.

In case the DISCOPAR^{EE} is deployed on the same device as the DISCOPAR^{DE}, the DISCOPAR^{EE} can interact with the DISCOPAR^{DE} to provide immediate feedback to the graph designer about the running program. For example, if the DISCOPAR^{DE} is used to create a mobile app for the citizen observatory, the live programming can be enabled to provide the graph designer a live preview of the mobile app's UI and test its functionality during its development.

In case the DISCOPAR^{EE} is deployed on a different device, the DISCOPAR^{DE} acts as a *distributed* live programming environment, where a graph designer can modify — from within its client-side browser — an application running on the server. In the current status, it is only possible to modify a running program on the server from within the client-side browser. In the future, we plan on adding support for real-time monitoring, such as data throughput of a process.

More use-cases of DISCOPAR^{DE} as live programming environment are discussed in chapter 5. The details on how this live programming is implemented are presented in chapter 6.

4.4 Facilitating End-User Programming

Section 3.7 introduced the most common goals of visual programming languages, i.e., making programming more understandable to some particular audience and improving the speed and correctness with which people perform programming tasks. These goals are directly applicable on the usability requirement of our CO meta-platform (cfr. section 3.3.1 - ULR2), as ICT-agnostic stakeholders must be able to correctly set up a citizen observatory without too much effort through the use of DISCOPAR. To achieve these goals, visual programming languages rely on four common strategies [14]: Concreteness, Directness, Explicitness, Immediate Visual Feedback. In the remainder of this section, we briefly introduce each of these strategies and explain how DISCOPAR adopts them.

Concreteness

Concreteness means expressing some aspects of a program using particular instances, i.e., using real values rather than a description of possible values [53]. Concreteness can be used in two ways: to provide feedback or to specify part or all of the program. For example, a WYSIWYG editor (“what you see is what you get”) automatically displays the effects of some portion of a program on a specific object or value.

DISCOPAR^{DE} is concrete by-design, as it enables programmers to specify what they want from the observatory through drag-and-drop actions where they manipulate specific instances of components.

The process configuration (cfr. section 4.3.3) is also an example of concreteness, as each process can be further customised by specifying actual values for a variety of configuration options. In case the live programming feature of DISCOPAR^{DE} is enabled, changes to these settings are automatically applied on the running application in the DISCOPAR^{EE}, potentially resulting in visual feedback in the live preview of the running application.

Directness

Directness is described by Hutchins et al. [58] as having two aspects: distance and engagement.

The first aspect of directness is a *small distance* between a goal and the actions required of the user to achieve this goal. A term closely related is the conceptual simplicity of a VPL [3], meaning that the underlying concepts are represented as naturally as possible and simplifies abstract concepts. It only emphasises logic which is directly pertinent to the application and not the programming mechanics. DISCOPAR

was created specifically to hide the technological complexity of a citizen observatory from the user. When creating a citizen observatory or campaign, users can create client-side visualisations from server-side data processing components without even realising they are setting up a distributed connection. Creating a (distributed) connection is as simple as drawing a line between an output port and an input port.

The second aspect of directness is a feeling of *direct engagement*, i.e., the feeling that one is directly manipulating the objects of interest. According to Shneiderman [116], the main principles for direct manipulation are: continuous representation of the objects of interest, physical actions or presses of labeled buttons instead of complex syntax, and rapid incremental reversible operations whose effect on the object of interest is immediately visible. Rather than relying on complex syntax, DISCOPAR^{DE} uses physical actions through its drag-and-drop interface and labelled buttons to directly manipulate a process, e.g., to configure a process, or delete it from the canvas.

Explicitness

Explicitness is the direct expression of certain aspects of semantics, without requiring the programmer to infer it. Visual programming languages usually shows relationships among objects, component, or modules in a very explicit way, for example, by drawing directed edges among related variables or statements.

Explicitness in DISCOPAR^{DE} is ensured as the connections between components are explicitly depicted to the graph designer. The connections indicate data flows from one process to another in the DISCOPAR^{EE}, without requiring the graph designer to infer it. Another example is the port typing mechanism. When dragging a connection from an output port, all the compatible input ports are explicitly highlighted.

Immediate Visual Feedback

Immediate Visual Feedback refers to the automatic display of effects of program edits. A closely related term is the *liveness* [124] of a program, which is the degree to which a visual programming language provides immediate feedback. Tanimoto proposed a four-level scale of liveness (see figure 4.14). The lowest level of liveness is the pure ‘informative’ level that uses the visual representation of the program as an aid to documenting or understanding the program, but not for implementing the program itself. The highest level is ‘informative, significant, responsive, and live’. It is a system that automatically provides incremental semantic feedback whenever the programmer performs an incremental program edit, and all affected on-screen values

are automatically redisplayed. Additionally, it also reacts to other events as well such as system clock ticks and mouse clicks over time.

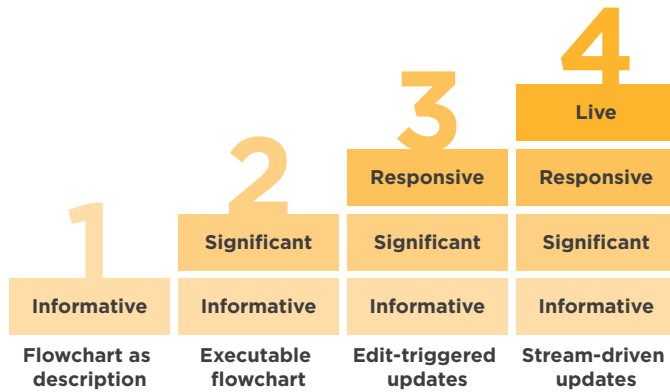


Figure 4.14: Levels of liveness in visual programming systems.

DISCOPAR^{DE} is designed to exhibit liveness at the highest level, which means that it is informative, significant, responsive, and live. It is informative and significant as the visual representation of the interconnected components is used to specify the actual implementation of the citizen observatory instead of just serving as documentation thereof. Responsiveness is present in DISCOPAR^{DE} as it displays the effects of changes made by the programmer automatically, that is, the programmer does not need to press a special button or do something in order to see the effects. For example, deleting a component automatically removes it from the canvas, in addition to any connection that was connected to the component. DISCOPAR^{DE} is live as the optional live preview (cfr. section 4.3.5) is capable of reacting to events such as the system clock, connection status, or microphone input, etc.

The advantage of implementing level four liveness for DISCOPAR^{DE} is that the amount and rate of feedback to the graph designers is maximised, which increases the correctness and speed at which they can program an observatory. Furthermore, immediately seeing the impact of incremental changes to the program makes it easier for graph designers to understand the effect of their actions.

4.5 Conclusion

This chapter introduced DISCOPAR, a new visual reactive flow-based domain-specific programming language. More specifically, we introduced the graph layer

Chapter 4: Language Concepts of DISCOPAR

of DISCOPAR, i.e., the visual syntax with which graph designers can construct programs. This graph layer is used by DISCOPAR^{DE}, a visual programming environment for DISCOPAR that is used throughout the citizen observatory meta-platform. We introduced the main concepts of flow-based programming and their visual representation in DISCOPAR's graph layer.

We presented DISCOPAR^{DE}, with key features including the input/output typing to ensure process compatibility, graph validation to prevent incorrect programs, process configuration to enhance customisability, and optional live programming mode to modify a running program, which may be deployed on the server. To ensure end-user usability and understanding, we adopted four common strategies into the design of DISCOPAR's graph layer: concreteness, directness, explicitness, and immediate visual feedback.

The concepts and visual syntax presented here serve as a guideline for the next chapter, which introduces the CO meta-platform and discusses how stakeholders can program a citizen observatory using DISCOPAR.

5

CONSTRUCTING CITIZEN OBSERVATORIES WITH DISCOPAR

DISCOPAR, presented in the previous chapter, is a visual reactive flow-based domain-specific language that lies at the foundation of our citizen observatory meta-platform. This CO meta-platform enables ICT-agnostic stakeholders and communities to construct their own citizen observatory and set up campaigns.

This chapter introduces the CO meta-platform. The first part of this chapter focusses on how a citizen observatory can be created through the use of DISCOPAR^{DE}. We describe how the mobile data collecting app, server-side data processing, and web-based visualisations of a citizen observatory are all implemented in DISCOPAR. The second part of this chapter explains how campaigns can be created within a specific citizen observatory. We conclude the chapter by describing the built-in calibration tool and the community component creator provided by the CO meta-platform.

5.1 Creating a New Citizen Observatory

The citizen observatory meta-platform enables stakeholders to instantiate their own CO on a particular topic, such as noise pollution, road conditions, etc. Creating a new citizen observatory is accomplished by specifying the name and purpose of the observatory, along with an image that will be used as logo throughout the observatory and as icon for the mobile app. The user who creates the CO is automatically assigned the role of citizen observatory administrator (cfr. section 2.4.1). Upon creating a new citizen observatory, the meta-platform automatically generates a series of dedicated web pages for that particular observatory, as depicted in fig. 5.1. Furthermore, the newly created CO is added to the list of observatories on the meta-platform's website. This website allows users to browse all currently existing citizen observatories and grants them access to each observatory's homepage.

The CO homepage acts as the central hub for that particular citizen observatory. The top part of the homepage consists of the observatory's logo and description, and presents a list of the various campaigns that have been deployed within that observatory. Each campaign is listed by its name, description, and the campaign's access level (cfr. section 5.2). A button enables any visitor to create his own campaign. The bottom part of the homepage shows a dashboard containing the visualisations specified in the Data Processing Design Interface, which is described in section 5.1.2. This dashboard can be used to show general statistics about the observatory, such as, for example, how many measurements have been uploaded in total, where they are made, etc.

The menu of the homepage changes depending on the user's access rights. The *CO administrator* has access to the following features:

- **Mobile App Design Interface (MADI)** is a web application that allows the CO administrator to create a customised mobile data gathering app for the observatory.
- **Data Processing Design Interface (DPDI)** is a web application that allows the CO administrator to specify server-side data processing and/or monitoring features of the citizen observatory.

The following features are accessible to *any* user:

- **Mobile Data Collection App** to contribute data to the citizen observatory. This app is created by the CO administrator through the MADI.
- **Observatory Data Analysis Interface (ODAI)** is a web application to analyse the citizen observatory's data.

DISCOPAR Observatories Instructions Tools Account Sign Out

Mobile App Design Data Processing Design Mobile Data Gathering App Observatory Data Analysis

NoiseTube
The NoiseTube mobile app extends the current usage of mobile phones by turning them into noise sensors enabling citizens to measure the sound exposure in their everyday environment. Furthermore each user can participate in creating a collective map of noise pollution by sharing geolocated measurement data with the NoiseTube community.

Campaigns + Create A Campaign

Radisson Blu	Noise mapping of Radisson Blu Hotel, Abu Dhabi	OPEN
Campaign: Highways into Abu Dhabi	Compare noise levels between major arterial roads into Abu Dhabi	REQUEST
Khalidiya Garden	Noise mapping of Khalidiya Garden, Abu Dhabi	OPEN

Dashboard

Daily Measurements

Sunday	335
Monday	353
Tuesday	234
Wednesday	371
Thursday	310
Friday	338
Saturday	227

CO Administrator

Mobile App Design Interface

Data Processing Design Interface

All Users

Mobile Data Gathering App

Data Analysis Interface

Figure 5.1: Automatically generated web pages of a citizen observatory.

5.1.1 Mobile App Design Interface

Each citizen observatory has exactly one specific mobile app that will be used for collecting data. This app is developed by the CO administrator using the Mobile App Design Interface (MADI), which is depicted in fig. 5.2.

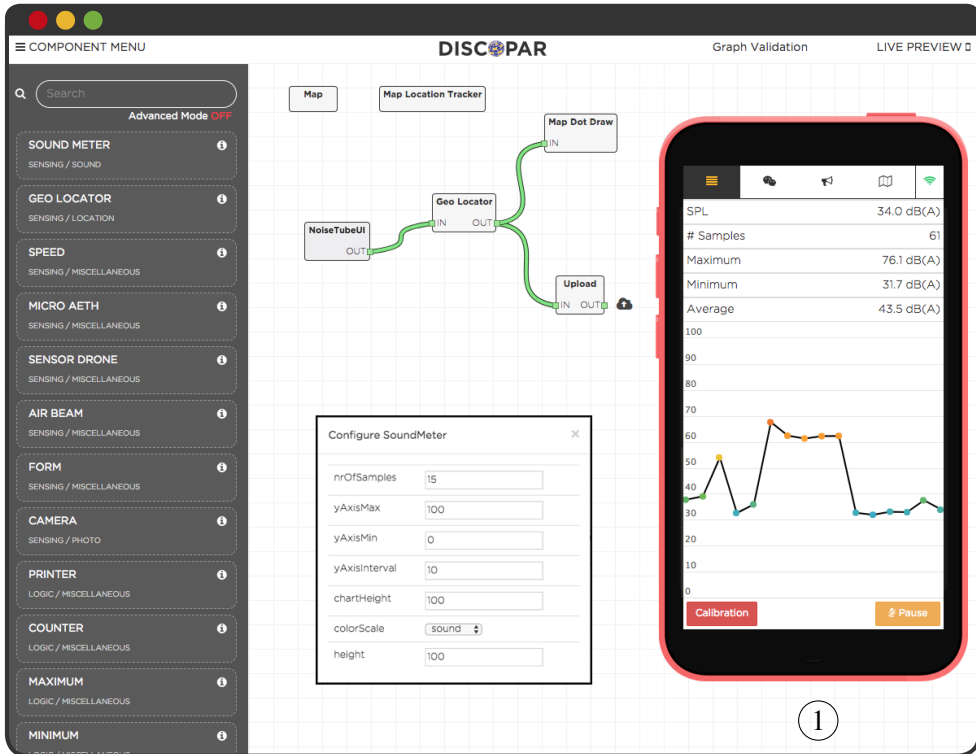


Figure 5.2: The Mobile App Design Interface, featuring the component library (left), designer canvas (middle), and a live preview of the mobile app (right).

The MADI uses DISCOPAR^{DE} and populates the component menu with a set of domain-specific components that the CO administrator can compose into a graph to implement the mobile app’s logic. The components that can be used by a mobile app can be classified based on their functionality:

- **Sensing components** produce data coming from smartphone sensors or external devices, or through user input from a participant.
- **Logic components** offer data processing capabilities such as filtering of incomplete data, and communication components to upload data to the observatory.

- **Visualisation components** provide the means to visualise the data generated by the mobile app.
- **Coordination components** handle the instructions sent out by campaign orchestration components to provide participant coordination on an individual level.

Within each of these categories, subcategories are used to further subdivide components into meaningful groups. For example, sensing components are further subdivided by the type of data they produce, such as sound levels, air pollution, etc.

The MADI enables DISCOPAR^{DE}'s feature to become a live programming environment (cfr. section 4.3.5). Behind the scenes, the mobile app's logic is executed by DISCOPAR^{EE} to provide the CO administrator with a live preview of the mobile app ①. As discussed in section 4.4, DISCOPAR features "level four" liveness. As a consequence, all the affected on-screen values and dependencies are automatically updated whenever the CO administrator performs an incremental program edit. The live preview of the mobile app also reacts to other events that the web browser provides, such as geo-location information. As a result, the CO administrator can more accurately predict the actual behaviour of the mobile app while it is being developed.

5.1.2 Data Processing Design Interface

Each citizen observatory is responsible to process, store, and visualise all the data collected by participants. The CO administrator specifies how this is done using the Data Processing Design Interface (DPDI), which is depicted in fig. 5.3. The goal of the DPDI is twofold:

First, the DPDI enables the specification of how all the data gathered by participants must be pre-processed *before* persistently storing and passing it to the campaigns deployed within the observatory. This pre-processing is particularly useful to apply data cleansing and transformation to uploaded data samples. For example, filtering out measurements without GPS coordinates, after which reverse geocoding is performed on the remaining data.

Second, the DPDI can be used to create visualisations for analysis and monitoring purposes. Visualisations can be public, in which case they are updated in real-time on the dashboard included on the observatory's homepage (cfr. fig. 5.1). Not every visualisation has to be made public, as the observatory administrator may want to keep certain statistics private.

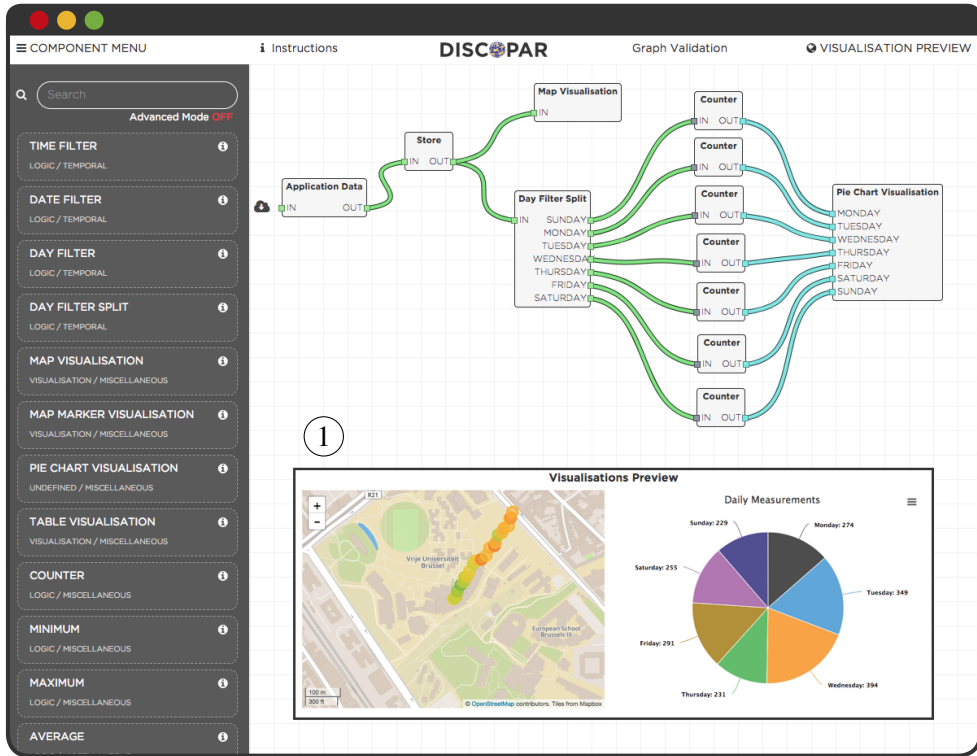


Figure 5.3: The Data Processing Design Interface, featuring the component library (left), designer canvas (top-right), and the visualisations preview window (bottom-right).

The Data Processing Design Interface is a good example of how DISCOPAR hides the non-essential complexity of creating distributed applications. The visualisation components specified in the DPDI, which are visible on the observatory’s dashboard, are web-based visualisations that are loaded every time the observatory’s homepage is loaded. However, these visualisations expect input from components that are deployed on the server. Therefore, behind the scenes distributed connections are established to ensure that these visualisations remain up to date in real-time. Details on the implementation of these distributed connections are provided in section 6.1.6.

Consider the example graph depicted in fig. 5.3. Although this graph does not perform any data pre-processing, it uses several components to produce data for two data visualisation components: a `MapVisualisation` shows the locations of the most recent data contributions, and a `PieChartVisualisation` compares the number of contributions made on each day of the week. These visualisations can be

previewed from within the DPDI through a dedicated window (cfr. fig. 5.3 ^①).

Similar to the MADI (cfr. section 5.1.1), the DPDI uses DISCOPAR^{DE} as a live programming environment. When a new observatory is created, a new DISCOPAR^{EE} is deployed on the server that executes the observatory's server-side logic. By default, new observatories only contain a component for persistently storing the data. Any change made in the DPDI to server-side logic is immediately mirrored by this DISCOPAR^{EE}. The CO administrator's actions on DISCOPAR^{DE}'s canvas thus modify the running program. For the client-side components present in the DPDI, i.e., the visualisation components, a separate DISCOPAR^{EE} is deployed on the client. This client-side DISCOPAR^{EE} interacts with the server-side logic to provide the CO administrator with a real-time live preview of the dashboard's visualisation during their development.

5.1.3 Mobile Data Collection App

The mobile data collection apps created through the MADI are web-based apps that can run in any modern mobile web-browser. This app initialises the DAG containing the app's logic as designed by the CO administrator. A citizen observatory's mobile app is responsible for measuring and monitoring the data, presenting the user with the necessary feedback, and — in case of temporary disconnections from the observatory — operating autonomously by buffering results until reconnection. Behind the scenes, the mobile app automatically establishes a two-way real-time connection to the observatory to upload data and receive feedback. Everyone who creates an account on the CO meta-platform is allowed to contribute data to a CO.

Each observatory's mobile app has a default layout, depicted in fig. 5.4, consisting of four tabs:

1. The main tab contains the actual mobile app UI.
2. The feedback tab displays messages from other users and general feedback coming from the observatory.
3. The campaign tab shows a list of campaigns deployed within the observatory whose intermediate results can be tracked in real-time.
4. The map tab depicts geographical information generated by the mobile app.

The campaign tab provides an overview of all the campaigns accessible to the participant. Clicking on a campaign loads their visualisations, enabling real-time monitoring of intermediate campaign results from within the app. Additionally, partici-

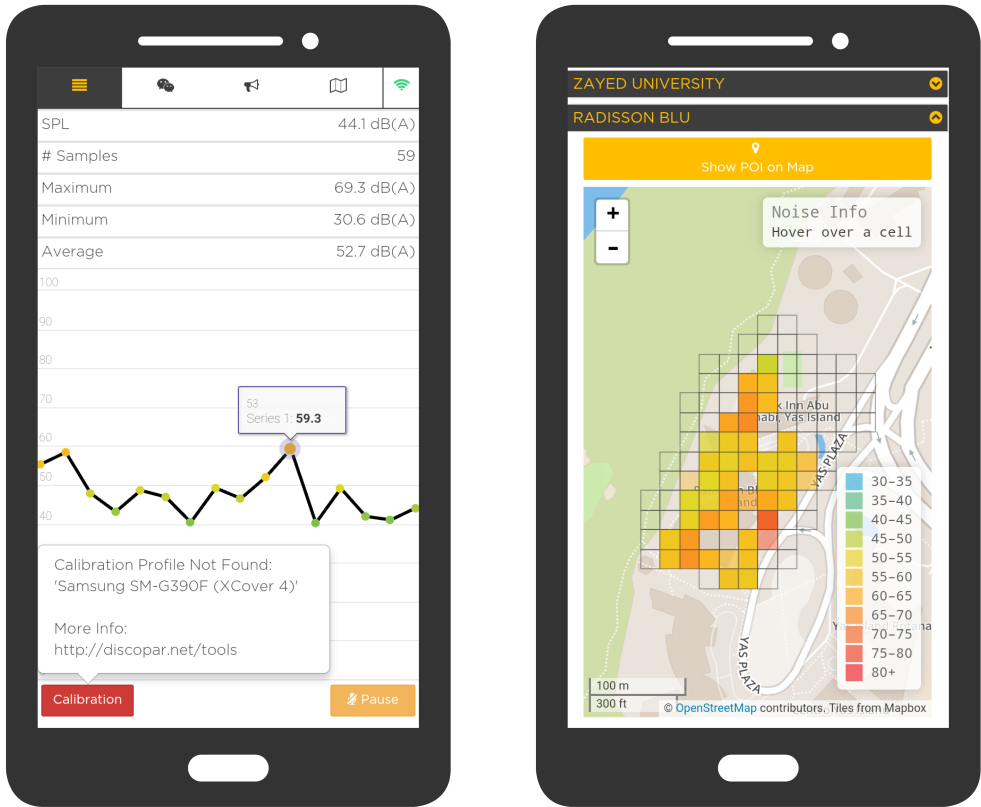


Figure 5.4: The Mobile Data Gathering App with the main tab (left) and the campaign tab showing intermediate results (right).

participants can toggle a button to indicate whether or not they want to receive campaign coordination messages by the campaigns orchestration components, if available for that particular campaign. We deliberately chose to not turn on any campaign-related features unless a participant explicitly chooses to do so. This way, we prevent participants from being overwhelmed by unwanted campaign coordination messages, while at the same time also reducing the bandwidth consumption of the mobile app. It also reduces the mobile app's CPU usage as it does not have to perform unnecessary updates to all of the campaign visualisations.

5.1.4 Observatory Data Analysis Interface

In addition to providing real-time data processing and visualisation, citizen observatories created through the CO meta-platform also support a more traditional query-like

approach. This is achieved through the Observatory Data Analysis Interface (ODAI) which enables end-users to query the observatory’s data for analysis and visualisation. For example, the ODAI of a citizen observatory regarding noise pollution could be used to see a particular city’s noise pollution.

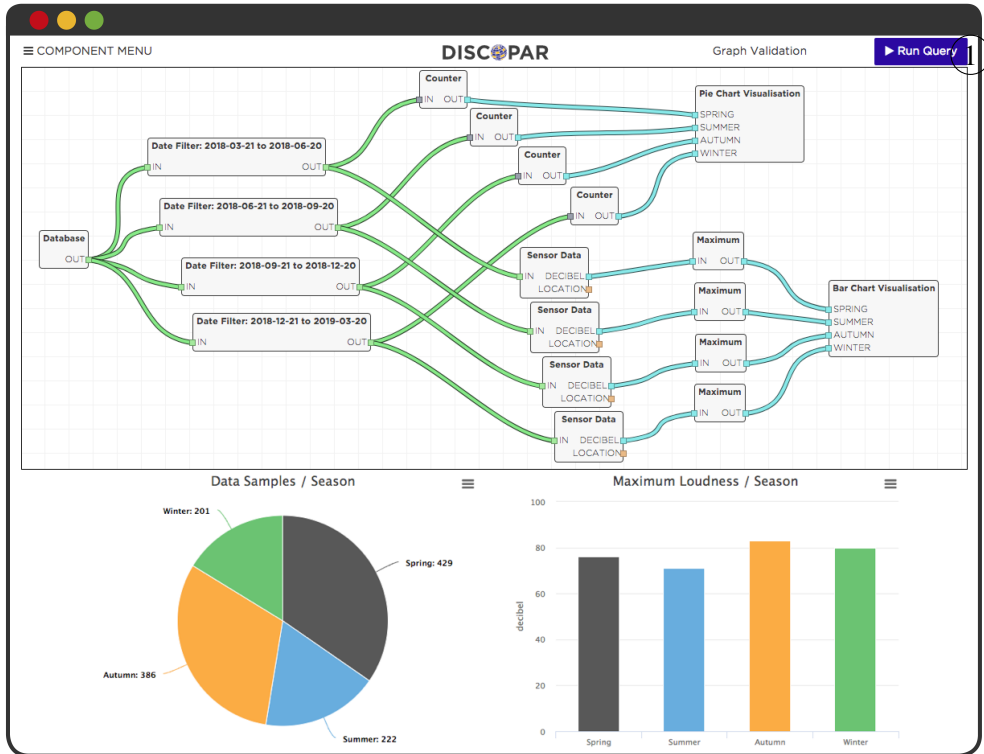


Figure 5.5: The Observatory Data Analysis Interface.

The ODAI, depicted in fig. 5.5 also uses DISCOPAR^{DE} and although very similar to the Data Processing Design Interface, it has a subtle but important difference: the graph’s source is not the Mobile App Data component that receives observations from each participant in real-time, but rather the Database component. Unlike the DPDI’s graph, which remains active during the entire citizen observatory’s lifetime, the ODAI’s graph is only executed once when the end-user decides to ‘run’ the graph (cfr. fig. 5.5 (1)). When this happens, all the observations are retrieved from the observatory’s database and provided as output produced by the Database component, after which they flow through the data analysis and visualisation components. In a way, end-users can thus program database queries by selecting and configuring data

filtering components. The raw data can also be downloaded in various formats for use in external data processing and visualisations tools.

5.2 Creating a Campaign

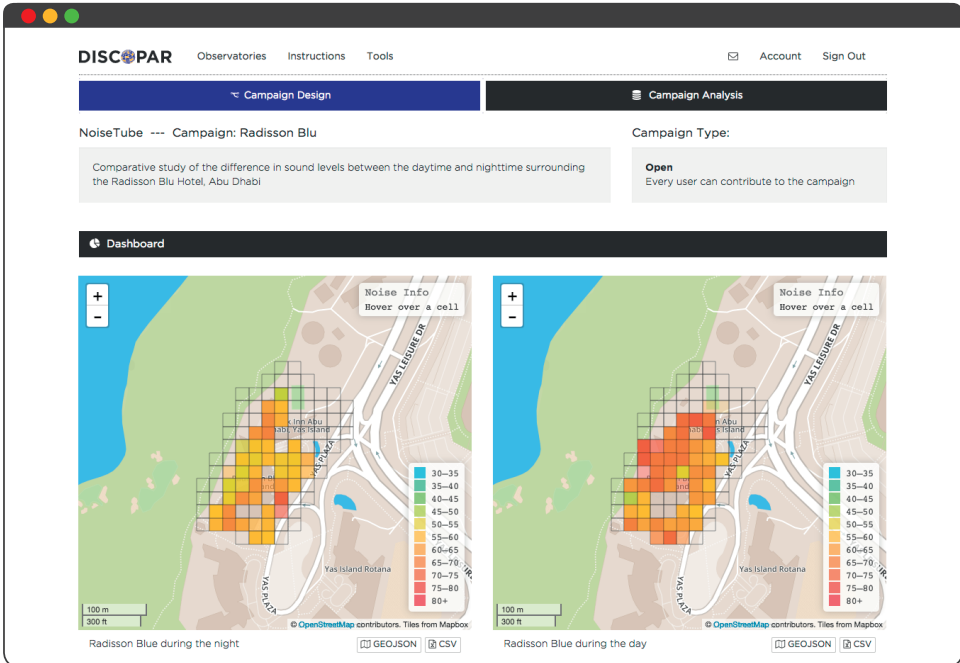
Each citizen observatory provides support to deploy campaigns. The goal of a campaign is to aggregate data for a particular goal or concern. Any user can deploy one or more campaigns in each of the citizen observatories that are available on the CO meta-platform. Users that create a campaign automatically take on the role of campaign administrator for that particular campaign. A new campaign is created by clicking the appropriate button on a CO's homepage and specifying the name and purpose of the campaign. Additionally, the campaign's access level has to be selected that specifies which users are allowed to participate in the campaign. Following access levels are supported:

- **Open:** Every user is allowed to participate. The campaign accepts observations from each mobile app user. The campaign is visible and accessible to anyone from the observatory's web-page.
- **Request:** Users request permission from the campaign administrator to contribute to the campaign. The campaign only accepts observations from users who are granted permission. The campaign is visible from the observatory's web-page, but not accessible without permission.
- **Invitation:** Users are invited by the campaign administrator to contribute to the campaign. The campaign only accepts observations from users who accepted the invitation. The campaign is only visible on the observatory's web page to users that accepted the invitation.

Creating a campaign results in the automatic generation of several web-pages, similarly to the creation of an observatory. These generated web pages are depicted in fig. 5.6. The campaign's homepage provides the campaign administrator access to:

- **Campaign Design Interface (CDI)** to specify the campaign protocol and visualisations.

Furthermore, the campaign's homepage provides every participant of the campaign (or every user in the case of a public campaign) access to:

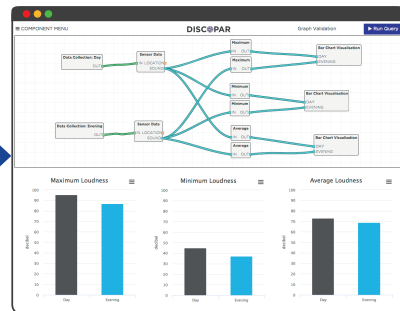


Campaign Administrator



Campaign Design Interface

All Users



Campaign Analysis Interface

Figure 5.6: Automatically generated web pages of a campaign.

- **Campaign Analysis Interface (CAI)** to offer a more traditional query-like analysis of campaign data.
- **Campaign Dashboard** to examine the real-time data visualisations.

The campaign dashboard is included in the campaign's homepage, as depicted in fig. 5.6. Visualisations defined in the CDI are shown automatically on the campaign's dashboard. These visualisations can either serve as reporting tools that enable the campaign administrator to monitor a running campaign, or as data visualisation for end-users to analyse the data from a campaign in real-time. As mentioned in section 5.1.3, these visualisations can also be loaded from within the observatory's mobile data gathering app where they serve as immediate visual feedback. These visualisations act as an incentive mechanism promoting further involvement of participants and end-users.

5.2.1 Campaign Design Interface

The Campaign Design Interface (CDI) enables the campaign administrator to specify the subset of data collected by the observatory that is of interest to the campaign. This is done by defining the campaign's protocol (cfr. section 2.3.1). Specifying the campaign protocol is done by placing constraints on the collected data (cfr. section 2.3.1) and the context in which the data collection took place (cfr. section 2.3.1). In addition to defining the campaign protocol, the CDI enables the campaign administrator to describe the campaign's desired output in terms of data collections, visualisations and maps.

The CDI, depicted in fig. 5.7, is a live programming environment where DISCOPAR^{DE} provides a view on the campaign's server-side logic and client-side dashboard visualisations simultaneously. Any changes made to the campaign's protocol are thus immediately applied on the server, and changes to the campaign's dashboard visualisations are immediately shown on the visualisation preview window (fig. 5.7 (1)), which has a similar purpose as the observatory's Data Processing Design Interface: it demonstrates what the current visualisations included in the graph look like.

The CDI also includes a geographical constraint editor (fig. 5.7 (2)) that enables the campaign administrator to define what areas are of interest to the campaign. Various forms of geographical constraints are supported, such as a simple point, a trajectory, complex polygon, etc. Notice however that specifying geographical constraints is not mandatory. For example, a campaign that collects data on eating habits of people from a certain age anywhere on earth does not require a geographical constraint.

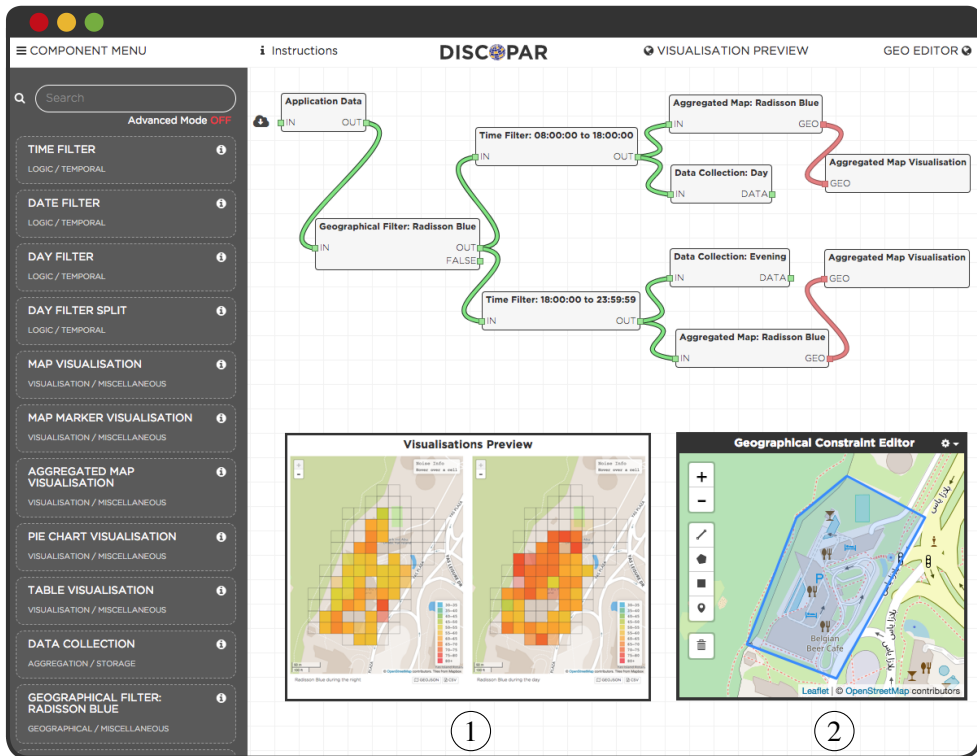


Figure 5.7: The Campaign Design Interface.

The geographical constraint editor is a good example of DISCOPAR’s dynamic component menu population (cfr. section 4.3.1): whenever the campaign administrator specifies a geographical constraint, a series of specific components for that geographical area are dynamically created and made available from the component menu. These include components to test whether or not an observation has been produced by an app while it is located in the specified geographical area, or a component that overlays the area with a grid layout and performs location-based statistical averaging.

The CDI’s component menu is categorised as follows:

- **Logic Components** provide various data manipulation components, as well as several filter components that can be used to specify the campaign protocol. These filter components are further divided into following sub-categories:
 - **Geographical components** can be used to add a geographical predicate P_A to the campaign’s protocol (cfr. section 2.3.1). In other words, they can be used to place a geographical boundary on the area of interest of a

campaign. Various geographical data filtering components are available, such as testing if a measurement was made inside a specified area or testing if the user stays on a predefined trajectory.

- **Temporal components** enable the campaign administrator to specify the temporal predicate P_T (cfr. section 2.3.1) to indicate when participants should be collecting data. These components filter data based on their time-stamp, which is added automatically to all observations that are produced by the mobile apps.
- **Contextual components** describe the desired context of the data collection. They define the contextual predicate P_C (cfr. section 2.3.1) of the campaign’s protocol. These components filter data based on meta-data such as the participant’s age or device model.
- **Aggregation components** collect and/or aggregate data so that it can be used in the visualisations depicted on the campaign’s dashboard. Data collections can also be used by end-users in the Campaign Analysis Interface to browse and analyse the data.
- **Coordination components** are capable of providing real-time coordination messages to participants. For example, location-based triggers can be used to prompt participants to gather data when they approach a particular area, or participants can be requested to move to a particular area that currently has a low data density.
- **Visualisation components** visualise (aggregated) data on the campaign’s dashboard. These visualisations are also available from within the mobile app’s campaign tab, enabling participants to monitor campaign progress in real-time.

5.2.2 Campaign Analysis Interface

End-users who want to browse and analyse the data that was collected in the context of an ongoing campaign can use the Campaign Analysis Interface (CAI) rather than relying on the visualisations depicted on the campaign dashboard. For example, consider a noise mapping campaign of a certain busy road during peak hours. At the end of the campaign, certain end-users may want to individually compare the different days of the week to see whether there is any notable difference. To this extent, the CAI provides a more traditional query-like approach, similar to the ODAI (cfr. section 5.1.4). The CAI thus enables campaign participants (or any user if the campaign is public) to run their own data analysis and visualisation.



Figure 5.8: The Campaign Analysis Interface.

The Campaign Analysis Interface (CAI), illustrated in Figure 5.8, enables its users to query the data collections made by the campaign. Data in these collections can then be further manipulated and filtered using DISCOPAR^{DE}. Similar to the ODAI, the graph constructed in the CAI is only executed once, rather than remaining active in the system indefinitely. When the graph is initialised, the data collections are retrieved from the database, after which they flow through the various data analysis and visualisation components. The raw data can also be downloaded in various formats for use in external data processing and visualisations tools.

5.3 Citizen Observatory Meta-Platform Tools

The CO meta-platform has a separate web-page where various additional useful tools for users can be presented. At the time of writing, the page provides access to the Sensor Calibration Tool and the Community Component Creator.

5.3.1 Sensor Calibration Tool

Sensor calibration is a comparative procedure where one smartphone’s sensor readings are compared with those of a trusted reference device that is considered to produce correct readings. Sensor calibration can reveal systematic errors that can then be corrected for in the future when collecting data with those sensors. Applying these corrections substantially increases the accuracy of smartphone sensors [41].

A major issue with sensor calibration is the vast number of different device models available. With new models being released on a regular basis, performing sensor calibration for each device model is tedious. Luckily, we observed from our previous expertise with the NoiseTube platform that there are many volunteers with the necessary expertise (such as noise pollution scientists) who are willing to perform this calibration themselves if they know their corrections will be integrated into the NoiseTube platform. While the NoiseTube platform relies on manual effort to integrate the calibration profiles, our CO meta-platform includes a Sensor Calibration Tool that enables stakeholders to upload their own calibration profiles, after which they are automatically integrated into the platform.

Galaxy S II	Sound Pressure Level (A)	Jesse	[[9,33],[13,7,38.5],[17,41.4],[20,5,44.5],[27,8,52.5],[35,7,59],[42,63],[45,68.2],[50,71.9],[53,73.4],[57,77],[58,80],[61,2,84.4],[65,85]]
Galaxy S5	Sound Pressure Level (A)	Andrew	[[15,21],[23,37],[35,56],[35,56],[39,64],[56,78],[78,90]]

Add Calibration Profile

Device Name

Sensor Type:

Mobile Value **Calibration Value**

Profile:

[21, 32]

Figure 5.9: The Sensor Calibration Tool.

The Sensor Calibration Tool, depicted in fig. 5.9, enables users to add their own so-called *calibration profiles*. Each calibration profile is linked to a specific device model and sensor. The calibration profile itself consist of a series of *calibration points*. A calibration point is a pair containing a reading of the smartphone sensor, combined with the corresponding output of the reference device. Calibration is performed by compensating the average measurement offsets of each device model through a correction term which is found by linear interpolation between calibration points for that

sensor. Calibration can be performed both on the client and server through the use of the calibration component.

In the current state, only calibration profiles for sound pressure levels using an A-weighting filter are included in the CO meta-platform. Observatories dealing with sound levels can thus utilise the calibration component to automatically calibrate data when a profile is available. Testing whether a calibration profile is available is done by extracting the meta-information from observations, which contains information indicating how the data sample was made, including the device model (cfr. section 4.2.5).

5.3.2 Community Component Creator

To improve the speed at which programs can be created, the CO meta-platform supports the concept of reusable and shareable end-user created components, named *Community Components*. These components can be created using the Community Component Creator (CCC), which is implemented through DISCOPAR^{DE}. The CCC enables the community to add new components by re-configuring and composing a number of existing components. Community components can be used throughout the platform. In a sense, they allow end-users to construct their own abstractions for mobile apps, data processing logic, and campaigns for other people to reuse. Community components can thus be compared to procedural abstraction of other general purpose languages. Each community component's internal implementation consist purely out of a DAG of other reusable components (which may be community components as well).

Figure 5.10 illustrates the Community Component Creator. A new component can be designed by visually programming its internal logic and defining the ports it exposes to send and receive data. The example in fig. 5.10 depicts a community component capable of filtering observations based on whether or not an observation was produced during the daytime in weekends. To do so, the community component relies on two internal components: a `TimeFilter` configured to daytime hours, and a `DayFilter` that is set up to only let observations pass that were produced in the weekend. A name and description can be specified when saving the community component, after which the component is published in the component library.

The community components mechanism ensures functional scalability of the CO meta-platform (cfr. section 3.3.2 - TR2), which is directly influenced by component's granularity. Community components enable the composition of fine-grained components that provide very basic features into larger, more complex components. This composition mechanism facilitates the addition of new features, as only a single fine-

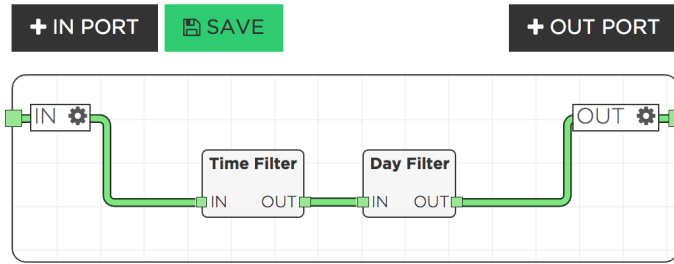


Figure 5.10: Community Component Creator.

grained component has to be implemented that can then be composed with various existing components to provide even more features.

Community components also increase the usability of the CO meta-platform (cfr. section 3.3.1 - UR2). Rather than having to reimplement every component composition from scratch, which is a time consuming effort, graph designers can use community components that already implement the desired feature. As such, the abstraction mechanism provided by community components has an impact on the efficiency at which end-users can program in DISCOPAR.

5.4 Conclusion

This chapter introduces our CO meta-platform that enables stakeholders to create their own citizen observatories. We present the various interfaces, implemented through the use of DISCOPAR^{DE}, that can be used to design each aspect of a citizen observatory and a campaign.

In chapter 7, we present three radically different citizen observatories to demonstrate the versatility of the CO meta-platform. However, we first provide some technological background in chapter 6 where we discuss implementations details of DISCOPAR and describe how the citizen observatories created through the CO meta-platform are hosted.

6

IMPLEMENTATION

For the sake of reproducibility, this chapter focuses on the implementation of DISCOPAR, our new visual reactive flow-based domain-specific language that lies at the basis of the citizen observatory meta-platform.

As is typical in flow-based programming (cfr. section 3.5), the design of DISCOPAR is split into two layers: the component layer and the graph layer. The latter is used in the implementation of DISCOPAR^{DE}, a web-based visual programming environment that is used throughout the CO meta-platform to design the various parts of a citizen observatory and a campaign. The visually composed programs are then loaded by DISCOPAR^{EE}, the execution engine implemented by DISCOPAR's component layer.

This chapter first introduces the various abstractions that together define the component layer in DISCOPAR. Next, we introduce the implementation of DISCOPAR's graph layer. Finally, we provide a high-level view on the underlying system architecture of the citizen observatory meta-platform and discuss the various technologies involved in every part of a CO's architecture.

6.1 DISCOPAR: Component Layer

Remember from section 3.5 that flow-based programming applications are represented as a directed graph consisting of processes as nodes and connections as edges. Processes access connections by means of *ports*. A process is an instance of a component, and runs concurrently with other processes.

The component layer of a flow-based programming language contains the source code of each component. In DISCOPAR, this source code is divided amongst several *packages*. Each package contains components about a certain topic. For example, the ‘geolocation’ package contains components offering various geolocation features, such as components interacting with the GPS sensor to provide information about the participants whereabouts or movement speed. An exhaustive list of DISCOPAR’s packages and their constituent components is presented in appendix A.

The component layer also included the necessary abstractions to create a component, spawn a process thereof, etc. Therefore, this section describes how FBP concepts such as components, processes, and ports are implemented in DISCOPAR’s component layer, which is written in JavaScript.

6.1.1 Components

Remember from section 3.5.2, which presents the characteristics of flow-based programming, that FBP is a form of component-based software engineering (CBSE) by design. CBSE defines components as software elements that conform to a *component model* and which can be independently deployed and composed according to a composition standard without modification [56]. The role of the component model is to define a set of standards for component implementation, interfaces, naming, meta-data, interoperability, customisation, composition, evolution, and deployment. We now introduce the component model of DISCOPAR, i.e., the set of standards each component in DISCOPAR must adhere to.

Component Implementation: Each component is implemented by instantiating the `Component` object, which requires two arguments: an object containing the component’s meta-data, and a function that contains the component’s black box logic.

Naming: Components in DISCOPAR require a unique name, as a component’s name is used to identify it in the component library. Within the same component, input ports must have a unique name. The same holds for output ports.

Meta-Data: The meta-data of a component contains its unique name. Optionally it contains the component's category, subcategory, description, and flags to indicate special cases such as components that can only operate on the client or server side. If no (sub)category is specified, it defaults to 'miscellaneous'.

Interoperability: Components are compatible if the output of one component can serve as input of another component (cfr. port typing section 4.2.4).

Customisation: Components can be customised in two ways: by providing them with a configuration object that acts as initial information packet (cfr. section 4.2.5), or by specifying user-customisable settings that can be configured by the graph designer through DISCOPAR^{DE}'s configuration window (cfr. section 4.3.3).

Composition: Components can be composed as long as their input and output ports are compatible. Client-side-only components cannot be included in a server-side application, and vice versa. Components with a special flag that indicates there can only be one instance of that component, can only be used once within the same application.

Evolution Support: Replacing a component with a newer version must not break the component's interface. This means that existing ports cannot be removed or renamed, although additional ports may be added. It is only allowed to modify the internal black-box logic of a component if it does not change the external behaviour of the component. The meta-information of a component, such as its description, can also be modified as long as it remains relevant with respect to the component's intended behaviour.

Deployment: Components can be deployed on any system where the DISCOPAR^{EE} is loaded. Restrictions from individual components, such as only allowing client-side deployment, still apply.

In the remainder of this section, we discuss how components can be implemented according to this model, and show how components can be initialised to create a process. Implementing a new component can be done in different ways: from scratch using JavaScript, through component composition, or a combination of the two.

Implementing a Component from Scratch

Implementing a new component is done by creating a new `Component` object. `Component` requires two arguments upon initialisation: The first argument is an object containing meta-data about the component, as described in the component model. The second argument is a function that contains instructions on how to transform a `DISCOPAR Process` into a particular instance of a specific component.

An example of a component implemented from scratch is shown in listing 6.1. This code extract contains the complete source code for the `Counter` component, which counts the number of information packets (cfr. section 4.2.5) it has received on its input port.

```
1 BasicComponents.Counter = new D.Component(  
2   {  
3     name: 'Counter',  
4     category: D.ComponentCategory.LOGIC,  
5     description: 'Counts and outputs the number of incoming data.'  
6   },  
7   function (process) {  
8  
9     // Register Ports  
10    process.addInPort('in', {}, increment);  
11    process.addOutPort('out', {type: D.DataType.NUMERIC});  
12  
13    // Component Black Box Logic  
14    var counter = 0;  
15  
16    function increment(_) {  
17      process.outPorts.out.send(++counter);  
18    }  
19  }  
20 );
```

Listing 6.1: Source code of `Counter` component.

The `Counter` component is added to the package `BasicComponents`. As a result, this component is automatically added to the `DISCOPARDE` component menu whenever the ‘basic’ package is loaded. Note that the various abstractions provided by `DISCOPAR`’s component layer, such as the `Component` object, are made available through the `DISCOPAR` or `D` global variable. The component’s meta-data is specified on lines 2 to 6, while lines 7 to 19 contain the component’s internal black-box logic to spawn a process of the `Counter` component. In case of the `Counter` component, a process must add an input port (line 10) that will execute the function `increment`

each time it receives an information packet. Additionally, line 11 adds an output port that outputs the status of the counter each time it is incremented. On line 14 the value of the counter is initialised. Lines 16 to 18 implement the `increment` function, which modifies the state by incrementing the process's internal counter variable, after which its new value is send to the output port.

Implementing a Component through Composition

Components can also be created through *component composition* [56]. Component composition is the combination of two or more components yielding a new compound component. This composition mechanism can thus be thought of as the FBP equivalent to procedural abstraction in procedural programming languages. The characteristics of the new component are determined by the components being combined and the way in which they are combined. The ability to create new components by using other components is essential to reduce complexity and maximise code reuse. Primitive components are components that do not contain other components. They are the basic building blocks of the system and are equivalent to built-in procedures in traditional programming languages.

The concept of community components, introduced in section 5.3.2, are one example of component composition (albeit on a high level of abstraction) where end-users program the internals of a new component through drag-and-drop actions in DISCOPAR^{DE}. A similar mechanism can also be used by component developers who wish to create new components on the textual source-code level by composing existing ones. For example, consider the `SoundLevelMeter` component shown in listing 6.2 whose implementation consist purely of other components (lines 7 to 9) which are interconnected (lines 11 to 12). Additionally, certain ports of the internal components are exposed (lines 14 to 16), which means that these ports act as input/output ports for the `SoundLevelMeter` component. The `destroy` function on (lines 18 to 21) is the equivalent of a destructor and is called when a compound component is deleted from the programming environment. it ensures that all the internally used components are properly disposed.

Spawning a Process from a Component

The components' source code listed in the previous examples only describe the internal black box logic of a `Counter` and `SoundLevelMeter` component. Spawning an actual process of a component is done by calling the `init()` function on the desired component stored in DISCOPAR's component library:

Chapter 6: Implementation

```
1 SoundComponents.SoundLevelMeter = new D.Component (
2   {
3     name: 'SoundLevelMeter',
4     ...
5   },
6   function (process) {
7     var spl = new D.Library.SoundPressureLevel.init ()
8     var calibration = new D.Library.CalibrateDecibel.init ();
9     var observationMaker = new D.Library.ObservationMaker.init ();
10
11     spl.outPorts.out.connect (observationCreator.inPorts.in);
12     observationMaker.outPorts.out.connect (calibration.inPorts.in);
13
14     process.inPorts.in = spl.inPorts.in;
15     process.outPorts.out = calibration.outPorts.out;
16     process.outPorts.decibel = calibration.outPorts.decibel;
17
18     process.destroy = function () {
19       spl.destroy ()
20       calibration.destroy ()
21       observationMaker.destroy ()
22     }
23   }
24 );
```

Listing 6.2: Source code of SoundLevelMeter component.

```
1 var counter = new D.Library.Counter.init (id, iip, settings)
```

Listing 6.3: Component initialisation.

The `init ()` function accepts three optional arguments:

1. An identifier to be assigned to the process. If none is provided, the system will automatically assign one to the process.
2. The initial information packet (cfr. section 4.2.5), i.e., a configuration object containing extra information to properly initialise a process. Intended to be used by component developers.
3. Initial values for the configurable settings that are made available in the component's configuration window of DISCOPAR^{DE} (cfr. section 4.3.3). Intended to be used by the graph designers.

Section 6.1.5 provides more details on the purpose of a process's identifier with respect to storing and establishes connections in a graph.

To further clarify the difference between a process's initial information packet (IIP) and settings, consider a citizen observatory where participants can use a mobile app to upload a geo-tagged photo combined with a rating (1-5 stars). The individual values of observations made for this observatory can be extracted through the use of the `ObservationData` component. This component accepts observations as input and outputs each individual value on one of its output ports, as illustrated by fig. 6.1.

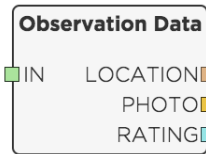


Figure 6.1: Visual representation of a `ObservationData` process.

The number of output ports of the `ObservationData` component is not defined in the component's source code. Instead, the number of output ports depends on what data is gathered and uploaded by the mobile application of the citizen observatory. This means that `ObservationData` cannot spawn a process without an IIP containing details on the data collected by the mobile app. This information is automatically extracted by analysing the graph representing the mobile app, using an approach similar to the graph validation presented in section 4.3.4, and stored as meta-information of the citizen observatory. An example of this meta-information is listed in listing 6.4. This meta-information is wrapped inside an object that acts as the IIP for the `ObservationData` component (line 12).

```

1 var iip = {
2   applicationmetadata : {
3     sensors : {
4       location: {type: "location"}
5     },
6     formFields : {
7       photo: {type: "file"},
8       rating: {type: "number"}
9     }
10  }
11 }
12 var obsData = new D.Library.ObservationData.init(null, iip, null)

```

Listing 6.4: Initialisation of component requiring configuration.

Note that the type of `ObservationData`'s output ports is automatically set based on the information provided by the IIP. Remember from section 4.2.4 that DISCOPAR assigns each port a type in order to ensure that components can only be

connected if they ‘understand’ each other, and that port types are visually represented in DISCOPAR^{DE} through the use of different colours.

In contrast, consider the `MapVisualisation` component that adds a map containing the individual observations to the observatory’s dashboard. The graph designer can customise processes of this component by specifying the dimensions of the visualisation, providing a short description explaining its purpose, and by selecting the tile layer for the map. Sample values for these settings are explicitly shown in listing 6.5. Providing settings to a component upon initialisation (line 8) will overwrite the default value of each setting.

```
1 var settings = {  
2   height: "400px",  
3   width: "100%",  
4   description: "Map depicting observatory data",  
5   layer: "openStreetMap"  
6 }  
7 }  
8 var m = new D.Library.MapVisualisation.init(null, null, settings)
```

Listing 6.5: Initialisation of component with settings.

Unlike the initial information packet, settings are customisable by the graph designer, enabling them to alter the behaviour of a process during its execution. These settings are displayed in the process’s configuration window (cfr. section 4.3.3) of DISCOPAR^{DE}, as depicted in fig. 6.2

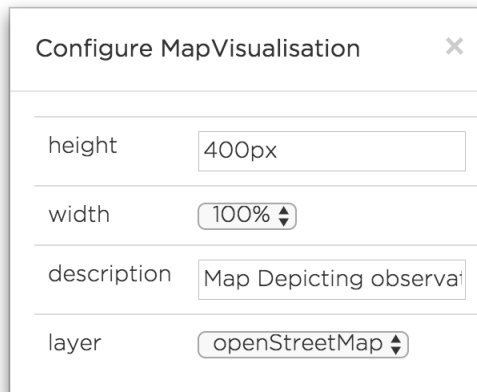


Figure 6.2: Configuration window of a `MapVisualisation` process.

6.1.2 Processes

Whenever a component is initialised from the library through the `init()` function, a new process is created (cfr.section 4.2.2). This newly created process then loads the black box logic as defined in the component's source code, which contains instructions on how to properly initialise a process for that particular component. To do so, a component's source code can utilise the `Process` object as illustrated in listing 6.6. Various meta-information fields, such as the process's unique identifier, are omitted for the sake of brevity

We now discuss each method supported by the `Process` object, as listed in listing 6.6.

```

1  function Process(component, iip, instanceSettings) {
2    var process = this;
3
4    process.addInPort = function (...) { ... };
5    process.addOutPort = function (...) { ... };
6    process.addSetting = function (...) { ... };
7    process.adoptSettings = function (...) { ... }
8    process.destroy = function () { ... };
9
10   return process;
11 }

```

Listing 6.6: Interface of the `Process` object.

The `addInPort` method adds an input port to a process. This method has four parameters:

- **portName:** string indicating the name of the port. Each component must have unique input port names.
- **configuration:** JSON object to specify various settings of the port. For example, `{type: D.DataType.NUMERIC}` indicates that the input port only accepts numerical data.
- **IPHandler:** (*optional*) function that is executed for each information packet received by the input port.
- **bufferHandler:** (*optional*) function that is executed after a series of information packets are received by the input port.

Chapter 6: Implementation

The `addOutPort` method adds an output port to a `Process`. This method has two parameters, i.e., `portName` and `configuration`, which serve the identical purpose as in `addInPort`.

The `addSetting` method adds a configurable setting to a process (cfr. section 4.3.3). `addSetting` accepts a name, initial value, callback and optional configuration. Every time the value of a setting changes, its corresponding callback is executed. Settings accept any value, unless the optional configuration object specifies a predefined set of values that restricts the set of values from which the setting must be chosen. For example, the `Map` component, which loads a map in the DOM, provides a setting to change the tile layer of the map. Rather than accepting any value, this setting must be set to one of the pre-defined supported tile layers.

The `adoptSettings` method enables a process that is implemented through component composition to adopt and expose the settings of internally used processes.

The `destroy` method is used to properly dispose processes, such as explicitly stopping underlying hardware signals used by the process. In the case of compound components, internally used processes have to be correctly destroyed as well. For example, the `destroy` function of the `SoundLevelComponent` (cfr. listing 6.2) disposes — amongst other — the internally used `SoundPressureLevel` process. Explicitly destroying the `SoundPressureLevel` is required to stop the media stream which captures sound samples from the audio input.

The `Process` object listed in listing 6.6 is used by default when initialising a component. However, certain components require additional features. For example, feedback components require access to the DOM, while community components (cfr. section 5.3.2) have an entire graph as their black box logic rather than built-in source code. As a result, different types of processes exist, which extend the `Process` interface or which override certain aspects of the internal implementation. These different types are indicated using an additional field in a component's meta-information, which in turn is used by a `Component`'s `init` method to decide what type of process is required to create an instance of the component. Currently, the following types of processes exist: `DOM-Process`, `Visualisation-Process` and `Community-Process`. We discuss each type in the remainder of this section.

DOM-Process is a process with access to the DOM. These processes are mainly used in the Mobile App Design Interface (cfr. section 5.1.1), as the mobile app's user interface consists of DOM-processes, ranging from simple data visualisations, such as displaying text on the screen, to more advanced ones, such as a line chart, pie chart, etc.

Each `DOM-Process` contains a field which holds the visualisation's HTML code. The `Process` API is extended with `addToDOM` and `removeFromDOM` methods, that add, or respectively remove, the process's HTML to the DOM. Additionally, an `addedToDOM` callback function is added, which can be used by processes that require additional set-up once its HTML is actually added to the DOM. A `resize` callback enables DOM-processes to update their visualisations in the case of a window resize, such as a participant rotating the smartphone from portrait to landscape.

Visualisation-Process is a special case of a `DOM-process`, and thus further extends the `DOM-process` interface. Visualisation processes are used by the web-based visualisation components that display observatory data or campaign data on a dashboard. Since they are designed to be depicted on a dashboard, each visualisation process provides several settings that can be configured to define the dimensions of the visualisations on the dashboard. Additionally, each visualisation supports various export options, enabling end-users to download the visualisations in various formats. Visualisations processes can specify which export options the support through the additional `loadExportOptions` method added to the `Process` interface.

Community-Process is an instance of DISCOPAR's community components (cfr. section 5.3.2). These user-defined processes consist of a graph describing the internal logic of the component. This graph, combined with the user-defined input and output ports of the community component, are persistently stored in the DISCOPAR community component database. When creating a process of a community component, its internal graph is loaded by the `loadInternalGraph` method, after which this graph is connected to the input/output ports of the community component through the `loadConnections` method.

6.1.3 Ports

Ports are the points of contact between processes and connections. In classical FBP (cfr. section 3.5.2), input ports provide *receive* functionality which consists of dequeuing information packets from a connection's buffer, and output ports provide *send* functionality to queue an information packet into the port of a connected process. In contrast, DISCOPAR is a reactive FBP language, meaning that processes operate using a publish/subscribe pattern, i.e., processes wait for data arriving on their input ports and publish data on their output ports. The actual sending of an information

packet is a normal JavaScript event that triggers the connected input port's callback function.

Input ports and output ports provide a different interface, although both inherit some common properties from the `Port` object shown in listing 6.7.

```
1 function Port(name, config) {  
2   var port = this;  
3   ...  
4   port.connections = {};  
5   port.distributedConnections = {};  
6   port.type = config.type || Globals.DataType.ALL.;  
7   port.subject = new Rx.Subject();  
8   ...  
9   return port;  
10 }
```

Listing 6.7: The `Port` object.

Each port must be assigned a name upon creation, and additional configuration settings can be provided by means of an optional second parameter. DISCOPAR has the convention that in case there is a single input port on a component, it should be named 'IN'. Similarly, in case there is only a single output port, it should be named 'OUT'. In case of multiple ports, meaningful names are preferred. For example, the `Accelerometer` component has three output ports, named X, Y, and Z, that output acceleration on the corresponding axis.

Each port keeps track of the connections attached to it. Each port makes a distinction between local connections to another process on the same device, and distributed client-server connections that connect to a remote process.

The type of a port (line 6) can be specified using the optional configuration parameter. Recall from section 4.2.4 that DISCOPAR prevents processes with different port types to be connected to one another, thereby ensuring that interconnected processes are always compatible. The exception to this rule is the `Any` datatype, which is assigned to a port by default if no type restriction is specified. Ports receive and send data through the use of a publish/subscribe pattern implemented using RxJS Subjects. This is explained below.

RxJS Subjects

The Reactive Extensions for JavaScript (RxJS) [125] is a set of libraries for composing asynchronous and event-based programs using observable sequences and fluent query operators. RxJS represents asynchronous data streams via the `Observable`

abstraction. `Observer` objects can subscribe to an `Observable`, which notifies the subscribed `Observer` instance whenever an event occurs inside the `Observable`.

A `Subject` is both an `Observer` and an `Observable`: it can subscribe to a data source, and acts as an observable for its own set of subscribed observers. In this way, the `Subject` can act as a proxy for a group of subscribers and a source. For example, `Subject` is typically used to implement a custom observable with caching, buffering, and time shifting. Additionally, it can be used to broadcast data to multiple subscribers. These properties make a `Subject` very suitable to implement `FBP` ports: on one hand, input ports are both listening to upstream information packets arriving from output ports, while at the same time producing a stream of that notifies the internal black box logic whenever new IPs arrive. This can either trigger an event handler function within the process or (in the case of a compound component) forward the data to the required internal processes. On the other hand, output ports can subscribe to data streams coming from diverse sources, such as smartphone sensors and hardware signals, and publish that data immediately as output of the process upon which input ports can subscribe.

Input Ports

The additional features provided by the `InPort` object are shown in listing 6.8. Method `subscribe` (lines 4 to 13) enables a process to react upon the arrival of new information packets by registering a callback function. The way `subscribe` handles IPs depends on whether or not a `bufferHandler` was specified in the `addInPort` call of `Process` (cfr. section 6.1.2). In case `bufferHandler` is omitted, `IPHandler` is registered as callback and thus executed for each IP individually (line 6). In case `bufferHandler` is defined, IPs arriving at an input port are grouped in a timed buffer (line 8) that collects IPs until the specified time has elapsed. The buffer time defaults to 10 milliseconds but can be overridden using the input port's configuration field `bufferTime`. Lines 9 to 12 subscribe a new callback to this buffered stream, which execute `IPHandler` for each IP in the buffer (line 10), in addition to calling `bufferHandler` on the buffer as a whole (line 11).

We conceived the buffered stream mechanism after observing a frequently recurring pattern within a component's source code: Several components contain a combination of fast operations that need to be executed for each IP individually, and slow, expensive operations that do not necessarily need to be executed for every single IP. Consider for example a visualisation component that displays the collected data from all participants on the observatory's web page. Each newly uploaded IP is

Chapter 6: Implementation

```
1 function InPort(name, conf) {
2   var inPort = new Port(name, conf);
3
4   inPort.subscribe = function (IPHandler, bufferHandler) {
5     if (!bufferHandler)
6       return port.stream.subscribe(IPHandler);
7     else
8       return port.stream.bufferTime(port.options.bufferTime)
9         .subscribe(function (buffer) {
10          buffer.forEach(IPHandler);
11          bufferHandler(buffer),
12        });
13   };
14
15   inPort.connectFromDistributed = function (...) {};
16   inPort.disconnectFromDistributed = function (...) {};
17
18   return inPort
19 }
```

Listing 6.8: The `InPort` object.

individually added to this component's state in real-time. However, actually updating and redrawing an element in the DOM is a costly operation, especially considering the potentially huge amount of data arriving at the same time. Therefore, it suffices to update the DOM only once for each buffer. Another example is a server-side database process, responsible for persistently storing each IP. Rather than inducing a write operation for each individual IP, this process only performs a write for the buffer as a whole.

The `inPort` object also features methods to connect and disconnect *from* distributed processes. These are explained in more detail in section 6.1.6.

By default, each input port is associated with a specific callback function which is only executed when IPs arrive at that particular port. However, DISCOPAR also provides support for more complicated scenarios requiring so-called synchronised input ports, i.e., ports which fire a single callback only when each individual input port has an IP in its buffer. Input ports can offer this synchronisation, in addition to a variety of other interesting stream operators, as a result of being built on top of RxJS.

Output Ports

A connection between processes always originates at an output port. The `OutPort` object (listing 6.9) provides the means to establish (and disrupt) a connection from an output port to an input port. `connect` (lines 4 to 7) establishes a connection between an output port and an input port. Since both types of ports are implemented on top of RxJS subjects, this connection can be established by subscribing the input port's `Subject`, taking on the role of `Observer`, to the output port's `Subject`, which takes on the role of `Observable`. The return value of RxJS's `subscribe` is a function that, when called, will cancel the subscription. Output ports keep track of each un-subscribe function for each of their outgoing connections (line 5). These outgoing connections are distinguished based on the ID of the input port they connect to.

```

1  function OutPort(name, conf) {
2    var outPort = new Port(name, conf);
3
4    outPort.connect = function (inPort) {
5      outPort.connections[inPort.id] = outPort.subject.subscribe(
6        inPort.subject);
7      outPort.onConnect(inPort)
8    }
9
10   outPort.disconnect = function (inPort) {
11     outPort.connections[inPort.id].unsubscribe();
12     outPort.onDisconnect(inPort)
13   };
14
15   outPort.send = function (data) {
16     outPort.subject.next(data)
17     for (var connectionID in port.distributedConnections) {
18       outPort.distributedConnections[connectionID].emit(connectionID
19         , data)
20     }
21   };
22
23   outPort.connectToDistributed = function (...){ ... }
24   outPort.disconnectToDistributed = function(...){ ... }
25
26   return outPort
27 };

```

Listing 6.9: The `OutPort` object.

Each time a connection is established to some process, the `onConnect` method is executed, which acts as an event callback that developers can use for a variety of purposes. For example, when a data collection process establishes a connection to a visualisation process, it can send its entire data in bulk to that particular visualisation process to bring it up-to-date. From that point on, the visualisation process will receive incremental updates from the data collection through the established connection.

The `disconnect` function (lines 9 to 12) cancels a particular connection by calling the correct `unsubscribe` function stored in the output port's `connections` field. Additionally, the `onDisconnect` callback is executed that developers can customise in a component's source code.

The `send` function (lines 14 to 19) enables a process to publish an IP to one of its output ports, triggering a response from the subscribed input ports of connected processes. Distributed processes cannot be directly subscribed to an output port through RxJS, as they have no `Subject` representing their input port on the same device. Therefore, the output port must send the data across each distributed connection, which are implemented in DISCOPAR through the use of web sockets. More details on how distributed connections are managed, including the use of the `connectToDistributed` and `disconnectToDistributed` methods, are provided in section 6.1.6.

Gate Ports

There are situations where a component's input port acts as input and output port at the same time. For example, consider the community component (cfr. section 5.3.2) depicted in fig. 6.3. This component's `IN` input port (1) acts as output port for the internally used components. Similarly, the community component's output port (2) acts as input port for the internally used components. As a result, these ports act as "gates" and provide the combined API of both input and output ports.

In future work (cfr. section 8.2.3), we plan on using gate ports to enable hierarchical grouping of components, i.e., reducing a group of components into a single component representing that group. This grouped component would then use gate ports to implement input ports and output ports for connections going to processes outside the group.

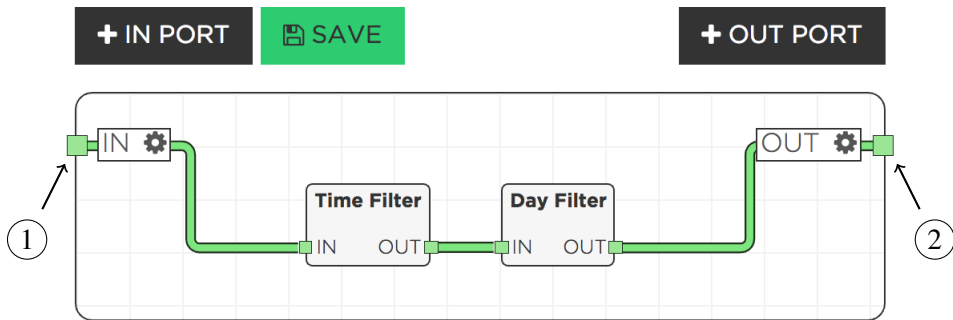


Figure 6.3: Gate ports of a community component.

6.1.4 Observations

Section 4.2.5 presents the concept of information packets and introduced a standard format for citizen observatory data, which DISCOPAR refers to as *observations*. Listing 6.10 depicts the implementation details of observations. By default, each observation stores the identifier of the participant that captured the data, in addition to the observatory's identifier for which the data is intended. Each observation automatically receives a timestamp on creation, in addition to the device model used by the participant. Location is an optional field, as not every citizen observatory needs to track where data was gathered (e.g., an observatory tracking eating patterns of people).

```

1 function Observation(user_id, observatory_id, data) {
2   var observation = this;
3
4   observation.user_id = user_id;
5   observation.observatory_id = observatory_id;
6   observation.datetime = new Date();
7   observation.location;
8   observation.device = DISCOPAR.Globals.getDeviceModel();
9   observation.data = data;
10
11  observation.addData = function(...) {...};
12
13  return observation
14 };

```

Listing 6.10: The Observation object.

The actual data is stored in the data field of an Observation, which is an object containing each collected data value. The addData method requires a data type and value.

6.1.5 Graphs

In section 4.2.6 we mention that the structure of an application in DISCOPAR is represented as a directed acyclic graph. However, cycles are often used as a means to store state in flow-based programming [17]. A self loop, for example, is an edge that loops back to an input port of the same process. This can be used by a process to pass along state data during subsequent activations. Because processes in DISCOPAR are stateful by design (cfr. section 4.2.2), cycles are not needed to store state, and in fact not allowed. This design choice was based on the assumption that ICT-agnostic users interacting with the DISCOPAR language find stateful processes more intuitive than having to manually program state using cycles.

The nodes of a DISCOPAR graph are processes, while the edges are the connections between ports. Before we present DISCOPAR's `Graph` object, we first discuss the textual notation of graphs. This JSON representation contains all the required information to initialise a graph and execute the program's logic. A graph in DISCOPAR is visually created through DISCOPAR^{DE}, from where its JSON representation is passed onto and loaded by DISCOPAR^{EE}. The JSON representation can also be used to persistently store graphs. As an example, consider the graph depicted in fig. 6.4 that calculates the average speed a participant is moving at, displaying this value on the screen as a simple label.

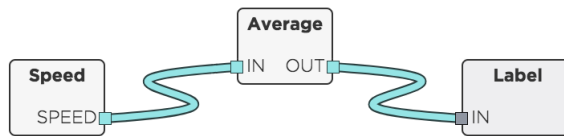


Figure 6.4: Simple example of a graph in DISCOPAR.

The JSON representation of this graph is shown in listing 6.11. Each JSON representation of a graph is comprised of five fields:

- Unique identification number of the graph.
- A collection of processes used in the graph.
- *(Optional)* A list of connections describing how the processes within the graph are interconnected.
- *(Optional)* A list of distributed connections, describing how processes connect to external processes.
- *(Optional)* Constraints imposed on the graph.

```

1 var graphJSON = {
2   id: "af886f62-bb39-4933-c539-831de2cb0513",
3   processes: {
4     b0b85c47-f96b-4ea9-aef0-1a89a8ea632c: {
5       name: "Speed"
6       iip: {},
7       settings: {},
8     }
9     66fd995f-c4a7-40ca-b534-a61119360346: {
10      name: "Average",
11      iip: {},
12      settings: {},
13    },
14    0d53b4a0-b577-401e-b4f2-4486a37b13ca: {
15      name: "Label",
16      iip: {},
17      settings: {
18        text: "Speed",
19        unit: "km/h"
20      }
21    }
22  },
23  connections: [
24    {
25      sourceProcess: "b0b85c47-f96b-4ea9-aef0-1a89a8ea632c",
26      sourcePort: "speed",
27      sourceGraph: "af886f62-bb39-4933-c539-831de2cb0513",
28      targetProcess: "66fd995f-c4a7-40ca-b534-a61119360346",
29      targetPort: "in",
30      targetGraph: "af886f62-bb39-4933-c539-831de2cb0513"
31    },
32    {
33      sourceProcess: "66fd995f-c4a7-40ca-b534-a61119360346",
34      sourcePort: "out",
35      sourceGraph: "af886f62-bb39-4933-c539-831de2cb0513",
36      targetProcess: "0d53b4a0-b577-401e-b4f2-4486a37b13ca",
37      targetPort: "in",
38      targetGraph: "af886f62-bb39-4933-c539-831de2cb0513"
39    }
40  ],
41  distributedConnections: [],
42  constraints: []
43 }

```

Listing 6.11: JSON representation of a Graph object.

The JSON representation of a graph stores processes based on their ID. Each process stores the name of the component that it is an instance of, in addition to the process's configuration and settings. In this example, only `Label` has configurable settings, as the label's text and optional unit can be customised.

Connections store the process ID, port name, and graph ID of both the output port (the source) and the input port (the target). This information is used to identify the ports and establish a connection between them.

Distributed connections store information similar to the local connections, in addition to extra information required to establish the remote connection. More details are provided in section 6.1.6.

Constraints can be imposed on the graph. These are used by the graph validation indicator (cfr. section 4.3.4) in order to test if the graph satisfies the imposed constraints. Each constraint has a type, and contains information on what parts of the graph the constraint is applicable to. An example of a constraint is shown in 6.12. This constraint forces the graph depicted in fig. 6.4 to have an incoming connection on the `in` input port of the `Label` process. Currently, three types of constraints are supported: `IncomingData`, `OutgoingData`, and `pathExists`. `IncomingData` verifies whether a certain input port of a process receives data, i.e., whether there is at least one connection arriving on the port. Similarly, `OutgoingData` checks if an output port has any outgoing connections. `pathExists` takes an input port of one process and an output port of another, and tests whether a path exists between these ports using the breadth-first search algorithm.

```
1 constraints : [  
2   {  
3     arguments: {  
4       port: "in",  
5       component: "0d53b4a0-b577-401e-b4f2-4486a37b13ca"  
6     },  
7     type: "incomingData"  
8   }  
9 ]
```

Listing 6.12: Example of a graph constraint.

Constraints are specified by the CO meta-platform itself, as they are included in the various graph templates used by the meta-platform. For example, each mobile app starts from a graph template including an `Upload` component that must have at least some incoming connections, otherwise data is never sent to the observatory.

```

1  function Graph(json) {
2    var graph = this;
3    ....
4    graph.fromJSON = function (json) {
5      graph.id = json.id
6      for (var id in json.processes) {
7        var jsonP = json.processes[id];
8        var process = Library[jsonP.name]
9          .init(id, jsonP.configuration, jsonP.settings);
10     graph.addProcess(process);
11     }
12     json.connections.forEach(graph.addConnection);
13     json.distributedConnections
14       .forEach(graph.addDistributedConnection);
15     graph.constraints = json.constraints;
16     return graph
17   };
18
19   graph.addProcess = function (process) {
20     graph.processes[process.id] = process
21     process.graphID = graph.id;
22   }
23
24   graph.addConnection = function (connection) {
25     sourceCmp.outPorts[connection.sourcePort]
26       .connect(targetCmp.inPorts[connection.targetPort]);
27     graph.connections.push(connection)
28   };
29
30   graph.addDistributedConnection = function(...){...}
31
32   graph.removeProcesses= function (...) {...};
33   graph.removeConnection = function (...) {...};
34   graph.removeDistributedConnection = function (...) {...};
35
36   graph.toJSON = function () {...};
37   graph.loadDOM = function (...) {...};
38
39   graph.validate = function () {
40     return graph.constraints.map(function (constraint) {
41       return Globals.GraphValidation[constraint.type](graph,
42         constraint.arguments);
43     })
44   };
45   ...
46   return graph
47 }

```

Listing 6.13: The Graph object.

A graph's JSON representation can be used to initialise a `Graph` object. The source code of the `Graph` object is displayed in listing 6.13. An “empty” `Graph` is initialised when no JSON representation is provided. Loading a graph from its JSON representation is accomplished by means of the `fromJSON` method (lines 4 to 17), which turns the JSON representation into a working program as follows: first, the processes included in the JSON representation are initialised. This is done by looking up the correct component in the DISCOPAR component library and spawning a process of it by calling the `init` function with the stored identifier, initial information packet, and settings specified in the JSON representation. These newly created processes are added to the graph object using `addProcess` (lines 19 to 22), which stores them by ID in the `processes` field. Next, `addConnection` (lines 24 to 28) establishes the connections between the various processes within the graph by calling `connect` on the appropriate ports. Similarly, `addDistributedConnection` initialises the distributed connections to processes residing on a different machine, whose details are described in section 6.1.6.

Certain graphs are simultaneously executed on multiple devices. For example, participants using the mobile app of a particular citizen observatory are executing the same graph on their devices. The processes of these graphs thus share the same ID, as their IDs are used to establish the correct connections.

The `toJSON` method transforms a graph back into its textual representation. Any changes made to the graph during its execution are included.

`loadDOM` takes a DOM container as argument and adds any process with a visual representation, such as a `DOM-Process` or `Visualisation-Process` (cfr. 6.1.2), to it. These types of processes have an additional configuration value indicating the order that they must be added to the DOM container, ensuring an identical layout than the one previewed during the graph's development.

`validate` (lines 39 to 43) performs the graph validation as discussed in section 4.3.4 by testing the constraints imposed on the graph and indicating whether or not they are violated. This method is called automatically whenever a change is made to the graph in DISCOPAR^{DE}.

DISCOPAR supports the dynamic modification of an already deployed graph, even if there already are information packets flowing through it. Processes and connections can be added, changed, and removed without stopping and restarting the graph. As a result, `Graph` also offers methods to remove certain parts of a graph (lines 32 to 34). More details of DISCOPAR's live programming environment are described in section 6.2.1

6.1.6 Distributed Connection Manager

Up until now, we have deferred the explanation on how DISCOPAR handles distributed connections, i.e., connections between ports that live on a different machine. In this section, we present the Distributed Connection Manager (DCM) and describe how ports can establish a connection to a remote port. A remote port is a port belonging to a process that resides on a different device.

The DCM is implemented using Socket.IO [25], a JavaScript library enabling real-time bidirectional event-based communication through webSockets. Socket.IO includes additional features such as auto-reconnection support as well as disconnection detection. It provides both server-side and client-side components with similar APIs such that both client and server can emit events and subscribe event listeners.

Since clients, such as participants using the mobile app and end-users consulting the web-based visualisations, can (dis)appear at any time, it is impossible for server-side processes to be aware of a new client-side process unless the latter explicitly notifies the server-side of its existence. Therefore, distributed connections are always initiated by the clients.

The first step in establishing a distributed connections is a client-side port sending a message to the DCM containing a request to be connected to a certain port on the server. This is done by calling an output port's `connectToDistributed` method, which is shown in listing 6.14. This method can only be executed on the client-side, and works as follows: first, a new `DistributedConnection` is created (line 2), which requires the port names, process IDs, and graph IDs, similar to a normal connection, in addition to information whether it is an outbound or inbound connection from the perspective of the client. Each `DistributedConnection` automatically generates an ID. More specifically, this ID is generated by concatenating of the output port's name and process ID, combined with the input port's name and process ID.

Next, `distributedConnectionSocket` is called (line 3), which either creates a new socket that automatically connects to the CO meta-platform's server, or it returns the already existing socket. This means that each client/server connection originating from the same client is handled by a single underlying web socket, resulting in less network overhead and, as a result, less bandwidth consumption. This optimisation was implemented to reduce the mobile data usage of participants.

Each output port keeps track of its outgoing distributed connections. Recall from listing 6.9 that an `OutPort`'s `send` method to also emits the data along each of its distributed connections.

Chapter 6: Implementation

```
1 outPort.connectToDistributed = function (port, process, graph){
2   var connection = new DistributedConnection(...)
3   var socket = distributedConnectionSocket();
4   outPort.distributedConnections[connection.id] = socket;
5
6   socket.emit('connectTo', connection);
7   socket.on('reconnect', function () {
8     socket.emit('connectTo', connection);
9   });
10  };
```

Listing 6.14: The `connectToDistributed` method of the `OutPort` object.

The output port then emits a `connectTo` event containing the newly created `DistributedConnection` (line 6). Because the server has no means to differentiate between a temporary disconnect and a permanent one, the output port will resend the request to establish a distributed connection whenever the socket reconnects to the server (lines 7 to 9).

On the server-side, the `connectTo` event is received by the DCM, thereby triggering the code shown in lines 1 to 7 in listing 6.15. Here, the input port to which a connection must be established is located by looking up the correct graph (stored on the server by ID), and then locating the correct process within that graph. Once the input port is found, it is subscribed to data arriving on the socket using that particular connection ID as event name. Recall from section 6.1.5 that certain graphs, such as the one implementing the mobile app logic, are executed on multiple devices simultaneously. The processes in each of these graphs share the same IDs and thus communicate across a distributed connection using the same connection ID. As a result, the server-side input port of a distributed connection automatically receives data from possibly multiple instances of its client-side output port.

Since each client is connected through a single socket, we use multiplexing to support multiple distributed connections using the same socket.

The `InPort`'s callback, which is defined in the component's source code, is automatically triggered by setting the next value of the input port's internally used `Subject` (line 6).

Ending a distributed connection is done by removing the listener on the corresponding connection ID, meaning that from that point on the server-side input port will no longer reacting to data arriving using that connection ID as event name.

```

1 socket.on('connectTo', function (connection) {
2   var graph = getGraph(connection.targetGraph)
3   var process = graph.components[connection.targetProcess]
4   var inPort = process.inPorts[connection.targetPort];
5
6   socket.on(connection.id, inPort.subject.next)
7 });
8
9 socket.on('disconnectTo', function (connection) {
10  socket.removeAllListeners(connection.id)
11 });

```

Listing 6.15: `connectTo` and `disconnectTo` event handlers of the DCM.

The code listed in listing 6.15 only demonstrates how a connection from a client-side output port is established to a server-side input port. Connections going from a server-side output port to a client-side input port are implemented similarly. In this case, the client-side input port send a `connectFrom` event to the DCM, which locates the corresponding server-side output port and requests that output port to establish a connection to the client-side input port. From that point on, the output port will also transmit data over the specified socket to the client. On the client, the input port is also listening for events using the connection ID as name.

By default, a process that sends data to one of its output ports results in that data being send to every input port connected it. A previously unmentioned feature that server-side output ports provide is the possibility to send data to only one particular client. This is particularly useful for orchestrating a campaign where individual participants receive personal coordination messages. Sending this personalised data is easily accomplished in DISCOPAR thanks to two reasons. First, observations, i.e., the information packets uploaded by participants, include the user ID of the participant (cfr. section 6.1.4). Second, server-side output ports keep track of outgoing distributed connections based on user ID. This is possible as each socket contains meta-information such as the user ID of the client that it is associated with. Using this information, DISCOPAR supports a `to` method that can be used from within processes as follows:

```

1 process.outPorts.out.to(observation.user_id).send(data)

```

Doing so overwrites the default `send` behaviour and only sends the provided data to the specified client.

6.2 DISCOPAR: Graph Layer

In flow-based programming, the graph layer provides a higher level of abstraction and enables one to reason about an application in terms of components and their connections without having to worry about the internal implementation of each component. The graph layer enables the graph designer to create a program by assembling various components in a directed acyclic graph. DISCOPAR's graph layer has a visual representation, the syntax of which was introduced in chapter 4. This visual graph layer is used by DISCOPAR^{DE} (cfr. section 4.3) to enable ICT-agnostic users of the CO meta-platform to design their own citizen observatory and campaigns. This section discusses some of the implementation details of DISCOPAR's graph layer as well as DISCOPAR^{DE}.

The graph layer of DISCOPAR is implemented as a standalone library, named DISCOPAR-UI, which extends DISCOPAR with functionality to visually represent processes. This distinction is made for efficiency reasons, as all the DISCOPAR-UI features do not have to be loaded when executing a graph. DISCOPAR-UI is implemented using a combination of web technology, namely Javascript, HTML, and CSS.

Drawing Components

Components are drawn on the screen by adding their visual representation to the canvas of DISCOPAR^{DE}. Creating the visual representation of a component is done by initialising a `UI-Component`, which takes a regular `Component` and adds several methods and meta-information related to visually represent the process on the screen. For example, a `UI-Component` contains the `draw` method, which creates a DOM node representing the component and applies appropriate CSS styling before adding it to the canvas.

Each component must also visualise its input and output ports. This is done in a similar fashion, i.e., by creating a `UI-Port` object that represents the port. `UI-Port` also features a `draw` method that creates a visual representation of a port by creating a DOM element (applying the correct colour based on the data type that the port accepts (cfr. section 4.2.4)), and by attaching that DOM element on the correct side of the graphical representation of the component.

A `UI-Component` also adds extra meta-information to a component such as the current X and Y coordinates of the component on the canvas. These are updated whenever the component is moved around on the canvas. Furthermore, this information is stored persistently, thereby ensuring that the component reappears in the same location on the canvas when re-loading DISCOPAR^{DE}.

Connecting Components

Connections are implemented using JsPlumb [67], a JavaScript library that provides the means to visually connect elements using Scalable Vector Graphics. JsPlumb provides various customisation hooks which were used to build DISCOPAR^{DE}'s visual feedback mechanism that highlights compatible input ports when dragging a connection from a certain output port (cfr. section 4.3.2).

Connections are coloured according to the type of the output port. Once a connection is attached to a compatible input port in the graph layer, a corresponding connection is made in the component layer of DISCOPAR. A connection can be removed either by clicking on it, or by detaching it from the input port.

6.2.1 DISCOPAR^{DE}

DISCOPAR^{DE} is the web-based visual programming environment of DISCOPAR. It can be integrated into any web page, as it only requires access to the DISCOPAR-UI library and a DOM element in which the canvas can be initialised.

DISCOPAR^{DE} enables users to program a distributed application: multiple (sub)graphs, possibly deployed on different devices, can be loaded into DISCOPAR^{DE} simultaneously. Components from each graph can freely establish connections amongst each other, thus creating a single distributed graph. As a result, a distinction is made between local connections (i.e., between two components on the same device) and distributed connections connecting components hosted on different devices. This is done as follows: upon creation, each graph is assigned to a particular device by the meta-platform. Since each component knows to which graph it belongs, it also (indirectly) knows on which device it will be deployed. Using this information, the correct type of connection is automatically created. Distributed connections are always added to the client-side graph, as it is the client who initiates the connection by design (cfr. section 6.1.6).

In the current status of our work, components added to the component menu are explicitly assigned to a graph, as illustrated in listing 6.16. This code snippet is part of the Campaign Design Interface (cfr. section 5.2.1), where the campaign designer can implement the campaign protocol (server-side data filtering) and data analysis (client-side web-based visualisations) simultaneously. Here, the DISCOPAR and DISCOPAR-UI library are loaded, after which two new graphs are initialised. The `processingGraph` will contain all server-side logic, while the `visualisationGraph` will contain the client-side logic. Processes from both these graphs can be interconnected in the visual programming environment.

Since campaigns may contain temporal filters, all the components related to temporal filtering are added to the component library with the indication that each instance thereof will be added to the `processingGraph`. Similarly, visualisations components added to the library will automatically add their processes to the `visualisationsGraph`.

```
1 require(["DisCoPar", "DisCoParUI"], function (D, DUI) {
2   var processingGraph = new DUI.UI-Graph(new D.Graph( ... ));
3   var visualisationGraph = new DUI.UI-Graph(new D.Graph( ... ));
4
5   D.Globals.TemporalComponents.forEach(function (cmp) {
6     DUI.AddComponentToMenu(cmp, processingGraph)
7   })
8
9   D.Globals.VisualisationComponents.forEach(function (cmp) {
10    DUI.AddComponentToMenu(cmp, visualisationGraph)
11  })
12  ...
13 })
```

Listing 6.16: Adding components to the component menu of DISCOPAR^{EE}.

Live Programming Environment

DISCOPAR^{DE} is synchronised in real-time, which means that users do not need to explicitly save their changes. Each action performed by the graph designer triggers a corresponding event which is immediately sent to the CO meta-platform using Socket.IO. For example, changing one of the settings in a component's configuration window will emit a `changeSetting` event containing the relevant information. These changes are then persisted on the server by updating the graph's JSON representation and storing it in the database.

If DISCOPAR^{DE}'s live programming mode is enabled (cfr. section 4.3.5) the changes are not only applied to the JSON representation of the graph, but also the running program, i.e., the in-memory `Graph` object. Therefore, DISCOPAR^{DE} is a *live* programming environment (cfr. section 4.3.5). When this graph is running on the server, users interacting with DISCOPAR^{DE} are performing *distributed* live programming: they are actively changing a program running on the server from within their web browser. The in-memory `Graph` is not redeployed when a change is made. This way, processes preserve their state when changes are made to the graph that do not affect them.

6.3 Citizen Observatory Meta-Platform Architecture

This section provides a high-level view of the system architecture of the citizen observatory meta-platform, and discusses the various technologies involved in each of the part of a citizen observatory. Figure 6.5 depicts the system architecture of a single citizen observatory as created by the CO meta-platform. This figure only depicts a single citizen observatory with a single campaign, but the meta-platform can run multiple citizen observatories simultaneously, each capable of hosting multiple campaigns and interacting with an unbounded number of mobile clients.

The mobile app, server-side processing, and web-based visualisations are each implemented using a different DISCOPAR graph. However, since these graphs are interconnected, a citizen observatory can be considered as one single distributed graph. DISCOPAR^{DE} supports the design of distributed graphs, which means that, in theory, every element of a citizen observatory could be implemented using a single interface. However, the fact that end-users, CO administrators, and campaign administrators have different access privileges, we deliberately chose to split up citizen observatory's graph along various interfaces, such as the Data Processing Design Interface, Campaign Design Interface, etc. Additionally, because of the limited scalability features of DISCOPAR^{DE} with respects to managing the screen real estate, programming the citizen observatory's distributed graph as a whole would quickly become tedious due to the large number of components depicted on the screen.

Every element of a citizen observatory is implemented in JavaScript. Although JavaScript was originally intended to be used in the web browser, it is now possible to program the server-side of an application using JavaScript as well thanks to Node.js [48]. In the remainder of this section, we provide more technical details about each element of a citizen observatory.

6.3.1 Mobile Data Collection App

Each observatory has its own dedicated mobile app. This is in contrast to reusable PS systems such as Epicollect (cfr. Section 2.2.2), where there exists only one mobile app that shows a list of all available projects that users can click on to participate.

Additionally, rather than having multiple codebases for each mobile operating system (iOS, Android, etc.), DISCOPAR uses web-based HTML5 apps that are platform-independent and only require a HTML5 compliant web browser to operate. Therefore, users do not need to download any native app via the app store and

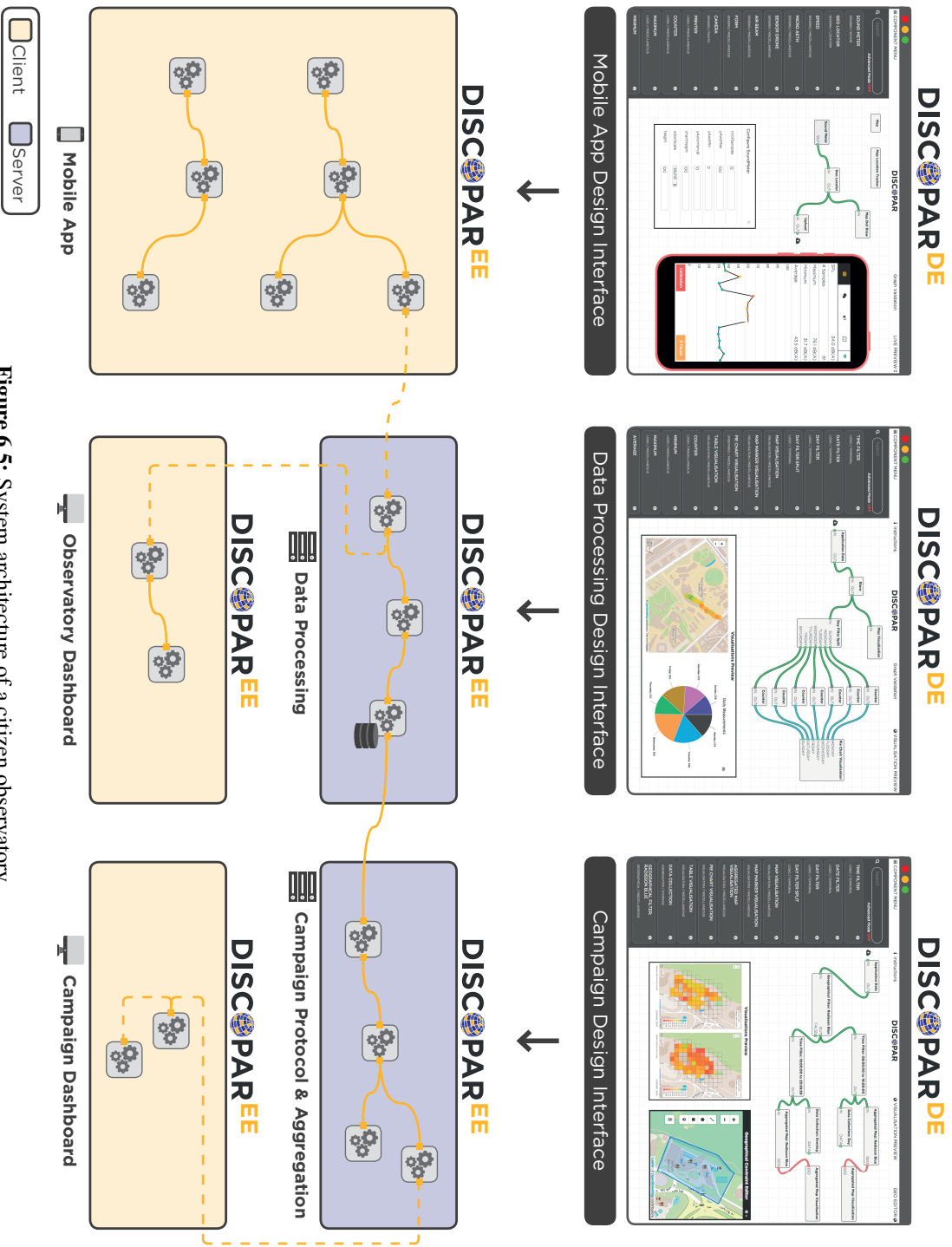


Figure 6.5: System architecture of a citizen observatory.

they can simply add the app's web page to their device's homescreen. Loading the web app automatically initialises the corresponding DISCOPAR graph representing the mobile app's logic. This means that users automatically use the most up to date version of the application. The observatory owner can modify the mobile app at any time, and users do not have to update the app through an app store.

Web-based apps are slowly gaining popularity over native apps since they are becoming increasingly powerful. The main advantages of web-based apps are:

- Common codebase across multiple mobile platforms.
- Developers — in our case ICT-agnostic stakeholders — do not need to submit their app to any app store for approval.
- Users do not need to bother with downloading the app from an app store. Additionally, a web-based app is automatically up to date.

These properties make web-based apps a good solution for our meta-platform where stakeholders want to develop and publish an app themselves. However, web-based apps also have some disadvantages:

- Performance is generally inferior compared to native apps.
- Limited access to the mobile device's hardware (e.g., sensors).
- Offline functionality are more difficult to support.

Although performance is less of an issue when dealing with mobile data collection apps as they are generally not very CPU-intensive. Limited device access is a more challenging issue for some data collection purposes. However, the gradual introduction of browser APIs already provide access to various device sensors, including accelerometer, ambient light detector, GPS, microphone, camera, etc.

Another challenge for web-based mobile apps is offline support. By default, web-based apps do not work when network connectivity is unavailable, as they are unable to download the app's logic. Solutions exist, such as Service Workers, which enable applications to take advantage of persistent background processing, including hooks to enable bootstrapping of web applications while offline. However, these technologies are not yet considered mainstream and therefore not yet supported by every browser.

To support more advanced features, such as offline support and additional device hardware access, an alternative approach such as a hybrid app can be used, which we discuss in more detail in section 8.2.

6.3.2 Server-Side Data Processing and Campaigns

The server-side contains both the observatory data processing graph, as well as the various campaign graphs. The server-side runs on a Node.js server. Since each citizen observatory can generate completely different types of data (noise measurements, mobility parameters, etc.), a flexible database system is required to persistently store observatory data. Therefore, we implemented the database using MongoDB, a NoSQL database supporting dynamic schemas. MongoDB use JSON documents in order to store records, just as tables and rows store records in a relational database.

Whenever a new citizen observatory is created through the CO meta-platform, a default observatory graph template containing some basic functionality, such as storing data in the database, is automatically deployed on the server. The observatory administrator can modify this graph via the Data Processing Design Interface (cfr. section 5.1.2) to add additional data processing features and visualisations to the observatory's dashboard. The `Store` component (which stores observations uploaded by participants in the database) is automatically connected to the `ApplicationData` component of each campaign running in the observatory (which acts as source for the campaign's graph). As a result, every campaign receives the pre-processed data upon which they can apply their campaign protocol.

Newly created campaigns also start from a default campaign template, similar to the observatory data processing graph. This graph is automatically deployed on the server, even though it provides no functionality apart from a source component that outputs all processed observatory data. The campaign can be implemented though the Campaign Design Interface (cfr. section 5.2.1).

Both the Observatory Data Analysis Interface (cfr. section 5.1.4) and Campaign Analysis Interface (cfr. section 5.2.1) deploy temporary graphs on the server. These graphs only remain active as long as the client is visiting the corresponding web page. The graph are initialised from their JSON representation, after which the observatory's data is retrieved from database. This data then flows through each process as defined in the Analysis Interface, after which it is automatically pushed across a distributed connection to the visualisation components deployed on the client.

6.3.3 Web-based Visualisations

The homepage and analysis interface of both a citizen observatory and a campaign include a dashboard containing web-based visualisation for end-users. These web-based visualisations are rendered on the client, and are implemented in HTML5, Javascript, and CSS. When loading these web-pages, the underlying visualisation

processes automatically connect themselves to the corresponding server-side output ports (cfr. section 6.1.6). From that point on, the visualisations receive real-time status updates from the observatory or campaign. When initialising their connection, the server-side components will first send their entire state to bring the visualisations up to date. From that point on, they will only receive incremental updates.

6.4 Conclusion

This chapter discusses the implementation of DISCOPAR's component layer and graph layer, and provided some technological details on the architecture of the CO meta-platform. We presented the component model that each component in DISCOPAR must adhere to, in addition to the source code of DISCOPAR's flow-based programming concepts such as components, processes, ports, and graphs. We presented the JSON representation of graphs which are the textual representation of graphs constructed in the graph layer. We also introduced the Distributed Connection Manager which enables real-time client-server communication between processes. We discussed how implementing citizen observatories entirely in JavaScript enables DISCOPAR's components to be deployed on the client and server, with the exception of some components, such as visualisations components that can only be deployed on the client side.

In the next chapter, we demonstrate the versatility of our CO meta-platform by creating different citizen observatories, and evaluate the usability of DISCOPAR^{DE} by performing two separate user studies.

7

DISCOPAR AT WORK

The previous two chapters discussed the features and implementation of our citizen observatory meta-platform, so that stakeholders can set up and configure the necessary ICT-tools of a citizen observatory and deploy campaigns without or with very little programming skills. In line with our research visions that consist of creating a reusable and reconfigurable approach for ICT-agnostic users to design and deploy citizen observatories (cfr. section 1.2), it is essential that we validate our solution in terms of expressiveness, suitability and usability through experiments in both laboratory as well as real-world conditions. This is the goal of this chapter.

7.1 Introduction

In chapter 2 we argued that despite the high societal demand, citizen observatory development remains labour-intensive, lengthy, and requires substantial technical expertise. This is mainly due to the fact that in current state of the art, each citizen observatory is developed from scratch, due to a lack of reusable and reconfigurable citizen observatory construction tools. As a solution, this dissertation focusses on conceiving a more generic approach. I.e., we present a CO meta-platform through which ICT-agnostic stakeholders are able to construct their own citizen observatories and design PS campaigns. This CO meta-platform was developed with four concrete research visions in mind (cfr. section 1.2). We now reiterate the research visions of this dissertation and explain how we will validate, in the remainder of this chapter, whether the CO meta-platform lives up to these visions.

Research Vision 1: Reconfigurable Citizen Observatory Platform Establishing a generic approach for building reusable and reconfigurable citizen observatories is by far the most complex and ambitious challenge that this dissertation tackles through the concept of CO meta-platform. To demonstrate the expressiveness of the CO meta-platform, we have created three radically different citizen observatories. These observatories each focus on a distinct domain, namely noise pollution, the quality of informal walking trails, and monitoring atmospheric conditions in terms of temperature, humidity, and atmospheric pressure.

Research Vision 2: Campaign Definition and Enactment Another challenge and responsibility of the CO meta-platform is to improve the chances for successful campaigning by automatically orchestrating the campaigns that are deployed in the observatories created on the platform. We therefore demonstrate the definition and successful enactment of several PS campaigns deploying in citizen observatories created through the CO meta-platform.

Research Vision 3: Reactive Citizen Observatories As explained in section 2.4.3, it is essential that participants receive immediate feedback to guarantee successful campaigning. We therefore highlight DISCOPAR's real-time campaign orchestration mechanisms that were used in real-world campaigns deployed within the observatories on noise pollution and quality of informal walking trails

Research Vision 4: ICT-Agnostic Usability Citizen observatory created through the meta-platform have to be configurable and usable by ICT-agnostic users. This vision is realised through the use of DISCOPAR^{DE}, which enables stakeholders and communities to design a CO and campaigns only by specifying their concerns and goals using domain-specific concepts. We validate the usability of DISCOPAR and the CO meta-platform as a whole through two end-user usability tests involving people without any programming knowledge.

7.2 Citizen Observatory Meta-Platform Expressiveness

In section 2.2, we introduced the concept of participatory sensing (PS), and categorised existing PS systems in two categories: Ad-hoc PS systems (section 2.2.1) and reusable PS systems (section 2.2.2). We discussed how most ad-hoc PS systems are only a proof-of-concept implementation to showcase the potential of PS as an alternative data gathering method for both sensorial data and user input. These systems

cannot be reconfigured and only work for the particular use-case they were designed for. Additionally, they provide no support for defining and enacting campaigns. On the other hand, several reusable PS systems were developed that solve the lack of re-configurability of ad-hoc PS systems. However, these reusable PS systems are limited in terms of the type of data that can be collected, as they focus only on discrete data, i.e., single-shot observations usually collected through questionnaires. They provide no support for continuous data streams originating from smartphone sensors. As with ad-hoc PS systems, campaign support is also largely missing.

The citizen observatory meta-platform presented in this dissertation provides the best of both worlds: The highly reconfigurable citizen observatories enable the creation of PS apps handling continuous data streams (e.g., sensorial data from both internal and external sensors), discrete data such as user input, or a combination of both. To showcase the CO meta-platform's expressiveness, we now present three radically different observatories that were created using the meta-platform.

7.2.1 NoiseTube^{2.0}: Observatory on Noise Pollution

The NoiseTube^{2.0} observatory built using our CO meta-platform is a re-creation of the existing NoiseTube platform (cfr. section 2.2.1). The NoiseTube^{2.0} CO was already demonstrated on various occasions in chapter 5, where it was used as a running example for the various tools that constitute the CO meta-platform. In this section we provide a more in-depth look at the components that were used to implement this citizen observatory, and we discuss various campaigns deployed in real-world scenarios.

Mobile Data Gathering App

The NoiseTube^{2.0} mobile data collecting app and its implementation are depicted in fig. 7.1. The mobile app is implemented in the Mobile App Design Interface using only 6 components.

The implementation is straightforward thanks to the abstractions provided by the component composition mechanism. If we look at the internal implementation of the NoiseTubeUI component, we notice that it is implemented using a composition of 12 components, as illustrated on the lower left side of fig. 7.1. Furthermore, one of these internally used components, i.e., the SoundLevelMeter component, is itself also implemented as a composition of 3 components (cfr. listing 6.2 in section 6.1.1). This clearly illustrates how component composition greatly reduces the size and complexity of graphs.

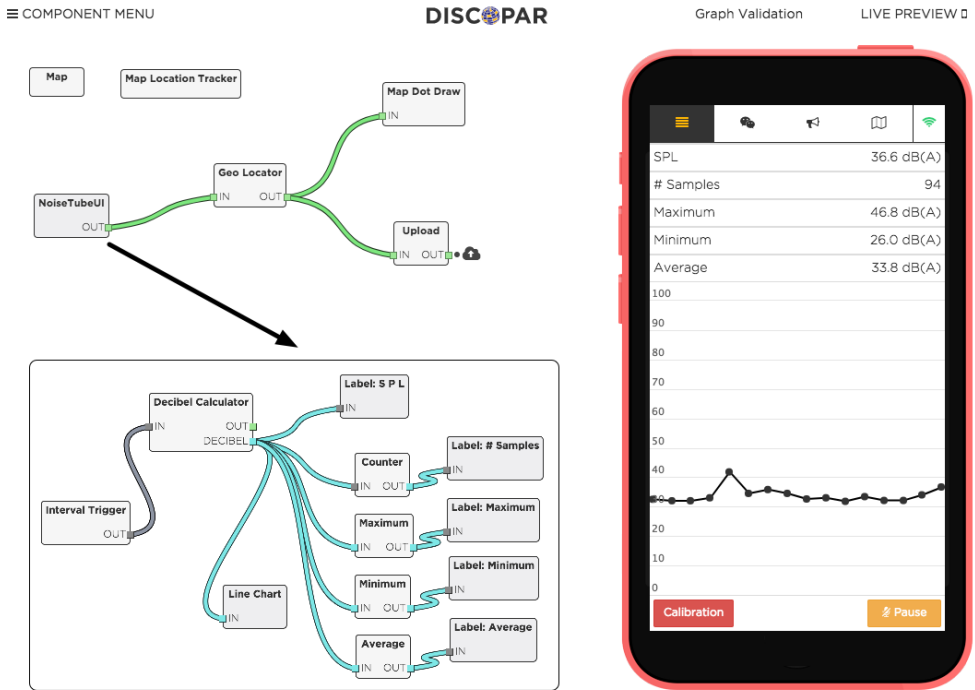


Figure 7.1: Implementation of NoiseTube^{2.0}'s mobile app in DISCOPAR.

Despite its straightforward implementation, The NoiseTube^{2.0} app is — with the exception of NoiseTube's noise source tagging feature — identical to the existing NoiseTube app. Both apps provide the means to collect sound pressure levels using the microphone, apply calibration when available, automatically tag observations with GPS coordinates, and provide feedback to users through visualisations.

Campaigns on Noise Pollution

The creation of NoiseTube^{2.0} demonstrates that our CO meta-platform is capable of building a mobile data gathering app that provides the same data gathering and visualisation features an existing ad-hoc PS system for measuring noise pollution. Where NoiseTube^{2.0} differentiates itself is the CO meta-platform's integrated support for organising campaigns, which is totally lacking from the existing NoiseTube system where campaigns are orchestrated using the manual effort of a domain-specialist.

We now provide a first demonstration of the suitability and expressiveness of the CO meta-platform for defining (noise mapping) campaigns. All of the following campaigns were created and executed during an international cooperation with

Zayed University in Abu Dhabi, United Arab Emirates. This cooperation was established upon invitation by Andrew Peplov, an associate professor in the department of Life and Environmental Sciences, who wanted to promote noise pollution awareness amongst his students by letting them participate in data gathering campaigns created through our CO meta-platform.

Campaign: Radisson Blue A first campaign was held to test the correctness of the CO meta-platform. The goal of this campaign was to measure the differences in noise pollution between daytime and night-time at the Radisson Blue hotel in Abu Dhabi. It is a good example of how the Campaign Design Interface (CDI) can be used to describe the campaign protocol (cfr. section 2.3.1). Figure 7.2 depicts the implementation of this campaign containing the campaign protocol, data aggregation components, and visualisations components that are visualised on the campaign’s dashboard.

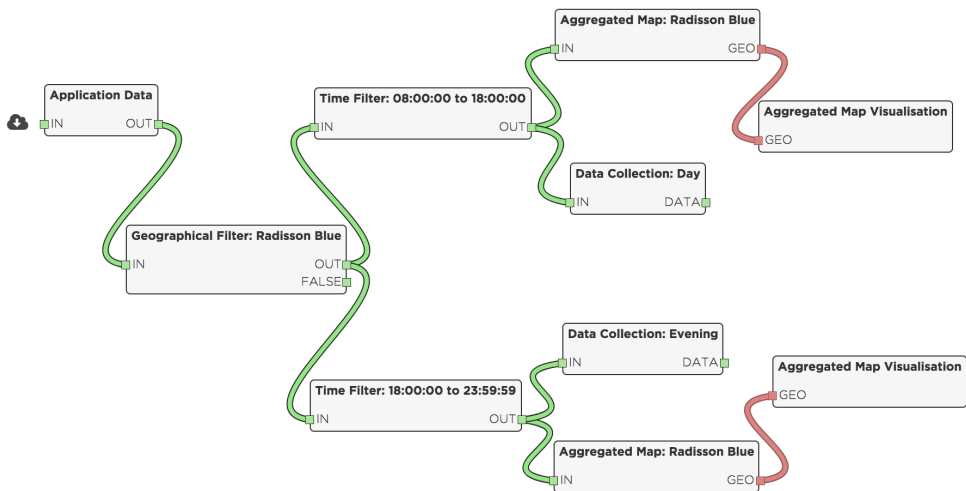


Figure 7.2: Radisson Blue Campaign Design.

This campaign’s protocol does not contain a contextual predicate (cfr. section 2.3.1). However, the measurement predicate (cfr. section 2.3.1) includes both a geographical (P_A) and temporal predicate (P_T).

P_A is implemented through the `GeographicalFilter` component, which filters out measurements made outside the vicinity of the hotel. This component was automatically generated and added to the component library of the CDI as a result

of indicating the area of interest in the geographical constraint editor, as illustrated in fig. 7.3.

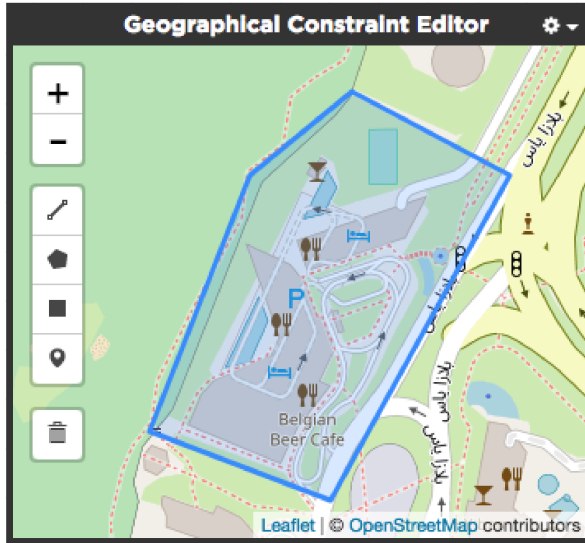


Figure 7.3: Geographical constraint editor of Campaign Design Interface.

P_T is implemented by means of two `TimeFilter` components. This campaign is only interested in measurements made between eight o'clock in the morning and midnight. Measurements made during this time interval are split into two groups in order to create two separate datasets, one that includes measurements made between 08:00–18:00 and another one that contains measurements made between 18:00–23:59. No additional temporal constraints are present, meaning that measurements from every month and day of the week are included in the campaign's two data sets.

The datasets created during this campaign can be used by the Campaign Analysis Interface (cfr. section 5.2.2), which enables end-users to create their own visualisations to browse and analyse campaign data. However, this campaign already provides some data visualisations on the campaign's dashboard: two `AggregatedMapVisualisation` components show a comparison of aggregated noise level during the day and evening, as illustrated in fig. 7.4. These components display the data as calculated by server-side `AggregatedMap` components. Each individual grid cell is coloured based on the average loudness levels. Clicking on a grid cell reveals additional information, such as the minimum and maximum of all the values recorded inside that cell.

7.2 Citizen Observatory Meta-Platform Expressiveness

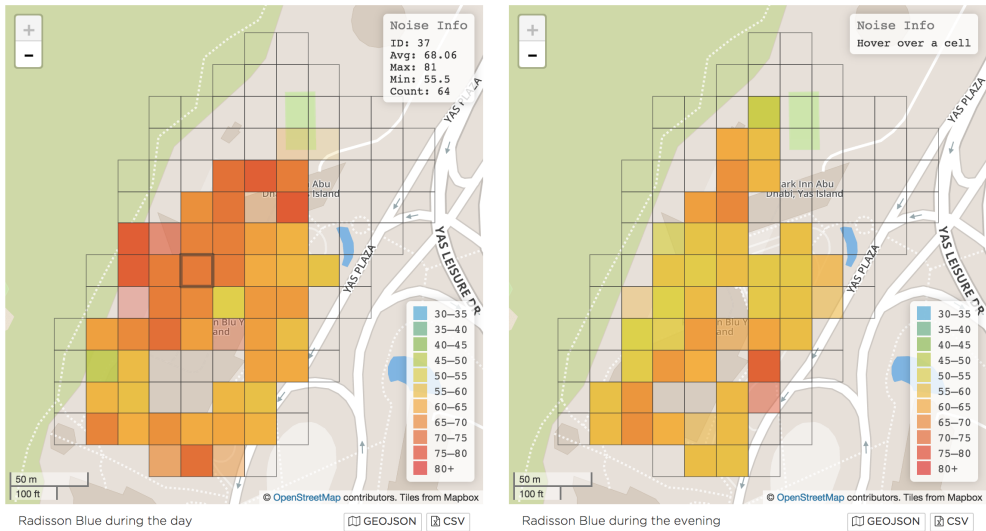


Figure 7.4: Visualisations on the ‘Radisson Blue’ campaign dashboard.

The quality of a campaign is inherently related to measurement density (cfr. section 2.3.2), as the enormous quantity of data that is typical for particularity sensing is usually translated into a more qualitative condensed representation. In this case, the condensed representation is obtained by location-based statistical averaging through the `AggregatedMap` component on the server. Sufficient data density, or the lack thereof, is made apparent by the `AggregatedMapVisualisation` component through the opacity of grid cells: Translucent grid cells indicate that the specified data density requirement is not yet obtained. Grid cells become increasingly opaque as density increases. Blank cells contain no data (yet). Because these visualisations are updated in real-time, the campaign administrator and visitors of the campaign web-page can monitor the campaign progress in real-time. Furthermore, participants can also load the campaign visualisations in the mobile app, enabling them to track the progress of a campaign in real-time and move to nearby locations that are still short of data.

The resulting noise maps of this campaign illustrate that there is a very clear increase in noise pollution during the daytime (left map of fig. 7.4), especially in the area surrounding the hotel’s swimming pool. Further analysis found that this was mainly caused by the children playing and shouting near the pool. This campaign thus demonstrates that the CO meta-platform is capable of enacting campaigns and producing the desired output.

Campaign: Zayed University A subsequent campaign using the NoiseTube^{2.0} citizen observatory was organised at the campus of Zayed University, where students were tasked to map the campus. In this case, the campaign was designed by professor Peplow, who took on the role of campaign administrator. He then tasked several students to look for loud area's on one side of the campus, and others to look for quiet area's. This campaign resulted in the aggregated noise map depicted in fig. 7.5. Both parking lots (north and south) are clearly identified as quiet area's, and the highest loudness values were measured near the university's technical installations for climate control.

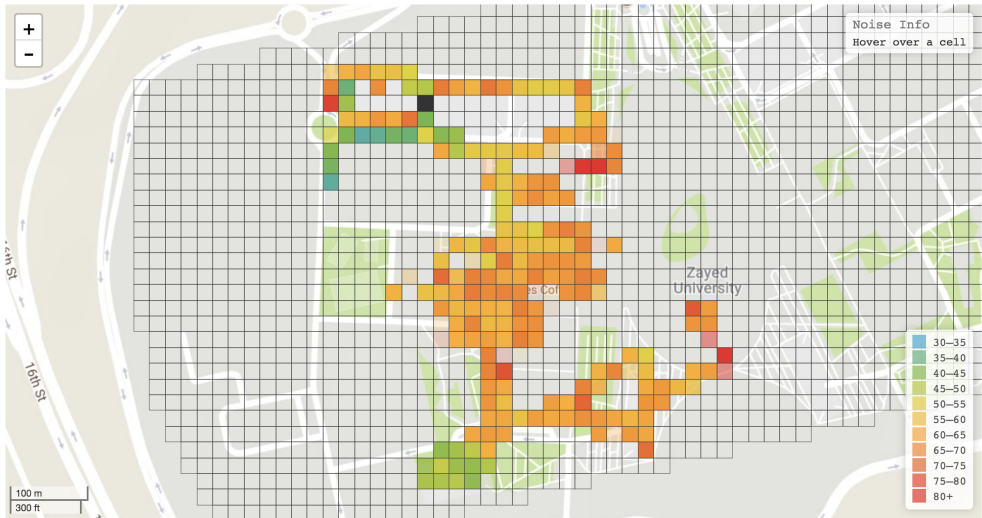


Figure 7.5: Visualisations on the ‘Zayed University’ campaign dashboard.

When paying close attention to the aggregated map, very low sound levels can be observed on the northern parking lot. The black grid cell indicates an unrealistically low value upon closer inspection. This happened due to the fact that one of the devices used to collect data was not calibrated, resulting in inaccurate data. The campaign administrator could have prevented this from happening by including a contextual predicate in the campaign protocol that only accepts measurements that originate from devices with a calibration profile.

Situations such as these are a good example of the convenience of the Campaign Analysis Interface (cfr. section 5.2.2). Here, the data set populated during the campaign's execution can be queried and sent through a new graph that does include a contextual predicate. The result is a similar map, albeit this time only including calibrated data.

Other Campaigns Several additional campaigns were organised in Abu Dhabi to create noise maps of various public parks and highways. We do not discuss each of these campaigns in detail as their campaign protocol is implemented using similar components as the aforementioned Radisson Blue and Zayed University campaigns. These campaigns were designed by professor Peplow himself by interacting with the CO meta-platform, without any involvement or assistance from us. For example, consider the campaign depicted in fig. 7.6. This rather complex graph implements a campaign that compares noise levels next to a busy highway during different times of the day. The successful implementation of this campaign can be considered a testimony that the CO meta-platform developed in this dissertation indeed enables ICT-agnostic stakeholders to create and execute their own campaigns successfully.

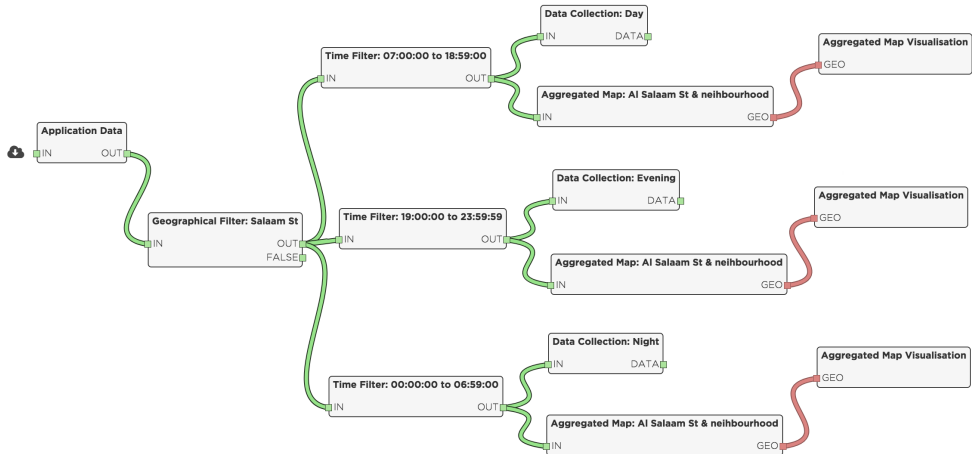


Figure 7.6: Implementation of the ‘Salaam St. and neighbourhood’ campaign.

Real-Time Coordination of Participants

In section 2.3.2 we defined the notion of a campaign. This definition states that a campaign is considered successful if the collection of measurements collected by the campaign is dense enough to generate a qualitative output. In the context of the aforementioned noise mapping campaigns, the “qualitative output” produced by these campaigns are aggregated noise maps, which can be considered dense if the every grid cell has reached a certain specified measurement density. From previous experience in organising campaigns for noise mapping, we observed that a threshold of 50 measurements per cell is sufficient to perform meaningful statistical averaging [41].

The noise maps generated by `AggregatedMapVisualisation` already provide visual feedback to participants in terms of grid cell density, giving them an idea on the campaign’s progress. However, the effectiveness of participants can be further improved by coordinating their data collection effort. Doing so results in each participant receiving individual instructions about where or when data contributions must be made. Campaigns can therefore rely on specialised coordination components that steer a participant’s movement based on a specified algorithm. However, finding an algorithm that offers the most optimal participant coordination in terms of coverage, effort, etc. is not within the scope of this dissertation, as this is an entire research topic on its own. We focus mainly on the technical difficulties in providing real-time individual feedback and coordination instructions to participants. To this end, we implemented a basic algorithm which always instructs a participant to the nearest incomplete campaign objective, which in this case are grid cells with insufficient data density. This functionality is implemented by the `FindNearestUncomplete` component, whose effect is depicted in fig. 7.7. In the future, we will investigate smarter coordination algorithms that can take multiple factors into account, such as the locations of other participants and their intended route.

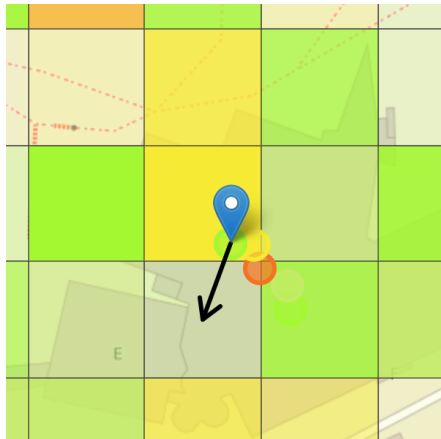


Figure 7.7: Instructing a participant to the nearest incomplete campaign objective.

When a participant activates the coordination mode on the mobile app’s campaign tab, he will initialise a new process that automatically establishes a distributed connection to the server-side `FindNearestUncomplete` to continuously send it location updates. The `FindNearestUncomplete` component also receives real-time updates of the campaign’s objectives, i.e., the aggregated noise map. Using this information, it calculates the nearest incomplete grid cell for each individual partic-

ipant, and sends this information back using a coordination message. Participants receive this message and update their map by adding an arrow indicating where the campaign instructed them to move to.

Note that the individual real-time updates on the aggregated map *also* take into account the contributions made by the other participants. It can thus be the case that the algorithm suddenly instructs a user to move somewhere else, as the current grid cell may have been completed by another participant.

The main problem using this coordination algorithm is caused by the way the grid cells are created: when a campaign wishes to make an aggregated map of a particular area, the entire area is covered using cells of a specified size. Unfortunately, many of these grid cells will cover inconvenient or inaccessible areas, such as areas covered with water, restricted areas, etc. As a result, there will always be cells that contain no data, resulting in participant receiving instructions to move to locations they cannot access. These inaccessible grid cells should therefore be manually removed. In the future, we plan on making these maps ‘smarter’ by taking into account topographical information.

7.2.2 Trage Wegen: Observatory on Pedestrian Experience

A second citizen observatory created using the CO meta-platform focusses on documenting the quality of informal walking trails. This citizen observatory was created in collaboration with Trage Wegen (Eng: “Slow Roads”), a Flemish non-profit organisation which has conceived the notion of a so-called slow road, i.e., a road intended for non-motorised traffic only. After 10 years of being active in this field, the organisation has succeeded to anchor this notion into road development plans of many municipalities in Belgium, while also building up a well-thought-out approach to inventarise and assess future development plans for slow roads. For efficiency reasons, they want to move away from their pen-and-paper data collection approach and are experimenting with the possibilities of mobile data collection apps. For this reason, we developed a mobile app for Trage Wegen using the CO meta-platform.

Mobile App

The implementation of the mobile data collection app is depicted in fig. 7.8. The main logic is provided by the `Form` component. Recall from section 4.3.3 that the default configuration window of a component can be modified to include a more advanced editor for component customisation. This is the case for the `Form` component which includes a drag-and-drop form builder. In the case of the Trage Wegen observatory, the `Form` component is configured to enable participants to evaluate the quality of

informal walking trails through the use of a photo, a rating, and an optional description. Both positive and negative aspects of selected points situated along a walking trail can be evaluated in this way. Each submitted survey is automatically geo-tagged through the GeoLocator component, after which it is uploaded to the server. Additionally, the documented locations are drawn on a map so the participant has an overview of the reports made.

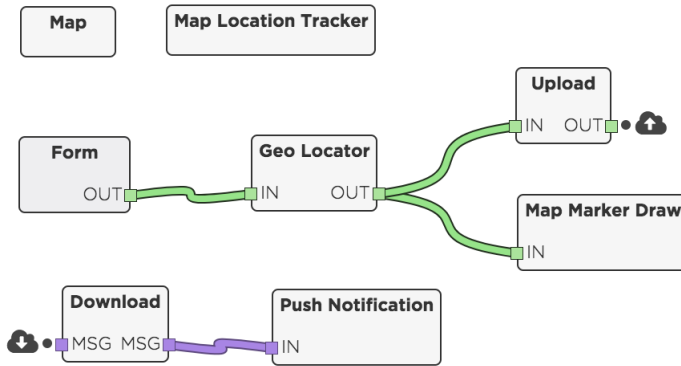


Figure 7.8: Implementation of the mobile app for the Trage Wegen observatory.

In section 5.1.3 we described how each mobile app created in the Mobile App Design Interface uses a default layout template. In case of the Trage Wegen observatory, the mobile app layout was changed upon request by the Trage Wegen organisation to be more in line with their own style. Using very minimal changes to the app layout’s code, a different layout was implemented, which is depicted in fig. 7.9. In this case, the main layout contained a map depicted the participant’s location. In the future, it is our goal to enable full customisation of the mobile app UI (cfr. section 8.2.3).

The Trage Wegen observatory was created specifically for use during the “Dag van de Trage Weg”, a yearly event where multiple organisations organise walks along various slow roads. Usually, these walks have a particular purpose, such as raising awareness of slow roads amongst the public. One particular campaign organised using this observatory was related to a walk organised in Anderlecht, one of the municipalities of the Brussels Capital Region.

Campaign: Anderlecht

The purpose of this campaign was to get feedback about the quality of the information displayed on specialised maps that are attached to various bus stops in Anderlecht. These maps illustrate the various points of interest and distance an average pedestrian

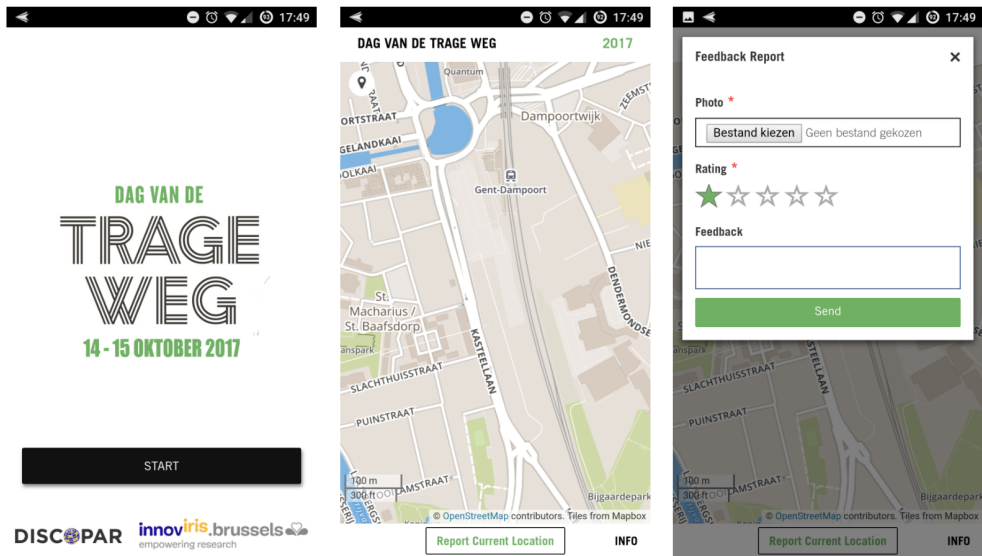




Figure 7.9: Mobile app of the Trage Wegen observatory.

can cover in 5 minutes, possibly through the use of a slow road. The goal of these maps is to inform commuters that walking via a slow road may be faster sometimes than waiting for the bus. Furthermore, participants were also encouraged to make reports about their findings of other aspects of the walk. For example, recycling bins placed in the middle of the walking trail, thereby blocking pedestrians (fig. 7.10). Six participants contributed to the campaign resulting in a total of 55 reports.



Figure 7.10: Participant reporting a negative element on the walking trail.

Time	Location	feedback	rating	photo
2017-10-15T11:58:30.016Z	50.83464, 4.30515	De map is geplaatst op een goede positie. Direct zichtbaar als jij uit het metro station kom.	5	
2017-10-15T12:09:57.037Z	50.83320, 4.30276	Kaart bushokje verzakt	3	
2017-10-15T12:11:05.784Z	50.83313, 4.30267	Plaatsing kaart is goed voor passage maar plastic voor map is verzakt en maakt kaart troebel dus moeilijk leesbaar	3	
2017-10-15T12:13:39.719Z	50.83285, 4.30142	Schoon!	5	
2017-10-15T12:14:02.685Z	50.83273, 4.30137	Mooie fontein met tuintje	5	

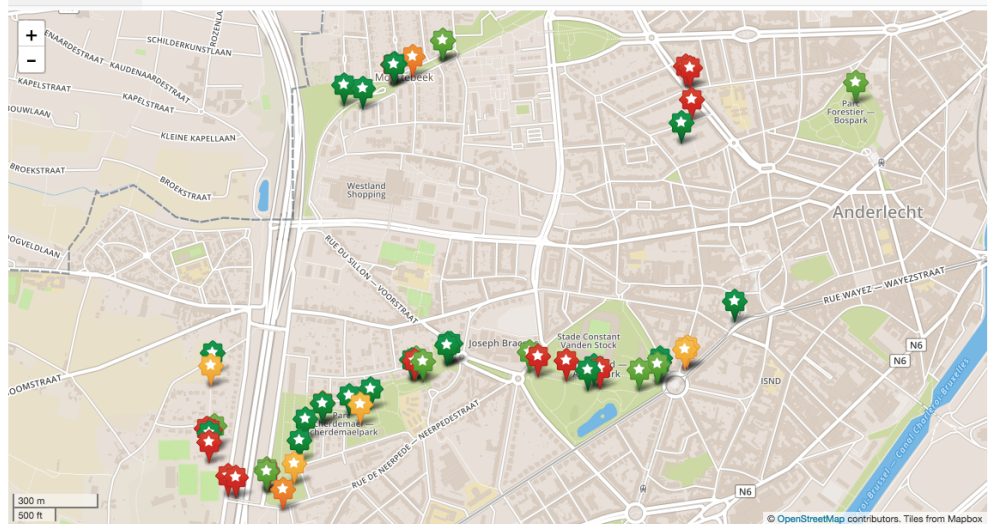


Figure 7.11: Dashboard of the Trage Wegen campaign in Anderlecht.

All the bus stops were added to the campaign through the use of the Campaign Design Interface’s geographical constraint editor (cfr. section 5.2.1). The geographical constraint editor provides support for adding elements in bulk rather than drawing them individually on a map. This was used to load the publicly available GeoJSON data of bus stop locations in Anderlecht.

Although the campaign was designed to collect any feedback on both the maps and other elements along the trail, a geographical predicate was used to create a separate data set for feedback related to the special maps attached to the bus stops. No other predicates were used in this campaign.

The campaign’s dashboard is depicted in fig. 7.11. The top pane is implemented through the `TableVisualisation` component, which creates a table and lists each uploaded measurement. The bottom is implemented through the use of a `MapMarkerVisualisation` component, which is configured to use the value of the rating for the marker’s colour.

Spatial Triggers

This campaign used spatial triggers as real-time participant coordination mechanism: whenever a participant got in the vicinity a bus stop, a notification was triggered on the participant’s mobile app prompting the participant to make a report on the nearby map. This mechanism prevented participants from missing a particular bus stop, and thus increased the data density and coverage of the campaign.

To do so, the app continuously uploaded each participant’s location to the campaign, where the `NearObjective` component perpetually tests whether a participant is within a certain distance of an “objective”, which in this campaign are the bus stops. The first time a participant is within the configured distance of an objective, an individual message is sent to that participant’s app which triggers the notification using the `PushNotification` component. These push notifications are an example of feedback being used as an incentive for participants to motivate them to increase their contribution.

7.2.3 SensorDrone: Observatory on Atmospheric Conditions

A third citizen observatory that was created using the CO meta-platform collects data of atmospheric conditions, particularly temperature, humidity, and atmospheric pressure. This observatory shows how mobile apps created through the CO meta-platform can also interact with external sensors over a Bluetooth connection.

The implementation of the mobile app is depicted in fig. 7.12. The bulk of the work is performed by the `Sensordrone` component, which is itself implemented

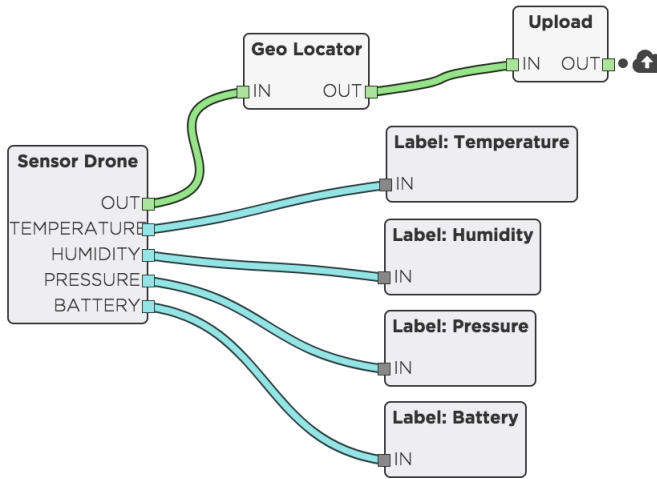


Figure 7.12: Implementation of Sensordrone mobile app.

using component composition. It relies on a component that is responsible for handling the Bluetooth communications, and another component that parses the data originating from a Sensordrone, a sensor fabricated by Sensorcon. Individual data values are sent to the appropriate `label` component, which updates the value on the mobile app's screen. The observation containing each data value is geo-tagged using the `GeoLocator` component, after which the observation is uploaded to the server. The resulting mobile app (depicted in fig. 7.13) visualises real-time sensor readings of temperature, pressure and humidity.



Figure 7.13: Sensordrone mobile app and external SensorDrone sensor.

Although this citizen observatory demonstrates the meta-platform's capability of interacting with external sensors (i.e., sensors that are not built-in into the mobile phone), there are some technical limitations to be taken into account due to the pure web-based implementation of mobile apps created through the CO meta-platform. More details regarding these limitations and a possible workaround are provided in section 8.2.1.

7.3 End-User Usability Tests

The CO meta-platform enables ICT-agnostic stakeholders to create their own citizen observatories and deploy campaigns. This is achieved by providing them with domain-specific concepts which can be visually assembled using DISCOPAR^{DE}. We validate the usability of DISCOPAR^{DE} and the CO meta-platform as a whole through two end-user usability tests that involve people without any programming knowledge:

- A first experiment focusses on the Mobile App Design Interface, where we tasked users to create their own mobile app using the CO meta-platform.
- A second experiment tests the usability of the Campaign Design Interface to verify whether users can design and deploy their own campaign.

To select an appropriate empirical study for these experiments, we took into account that the CO meta-platform developed in this dissertation is a new tool which is still a prototype. Therefore, the user base is virtually non-existent. Additionally, time restrictions prevented us from studying participants over a longer period of time. Given these restrictions, the most reliable and feasible type of experiment is a *quasi-experiment* [16]. A quasi-experiment is an empirical study which lacks random assignment of the groups or other factors being studied. A quasi-experiment, as opposed to a scientific experiment, does not allow us to make any founded claim regarding the usability of CO meta-platform. Nevertheless, it does provide insights about how the platform's intended end-users perceive and value the features of a visually programmable citizen observatory, in addition to providing anecdotal evidence on the usability of the CO meta-platform by ICT-agnostic users.

7.3.1 Quasi-Experiment: Mobile App Design Interface

A first quasi-experiment tested the Mobile App Design Interface of the CO meta-platform in order to study the intuitiveness and practical use of its features. This

experiment was performed during the international cooperation with Zayed University, involving 9 female bachelor students in Environmental Science, with participant ages ranging between 19 and 27. The curriculum of this bachelor program does not contain any programming courses, and we checked that none of the students had a background in ICT. Furthermore, in order to properly test how intuitive the UI is, they did not receive any explanation on how to use DISCOPAR^{DE}.

The students were tasked to create their own mobile app that is capable of measuring sound pressure levels (similar to the NoiseTube^{2.0} mobile app presented in section 7.2.1), using the Mobile App Design Interface (cfr. section 5.1.3). In order to be considered successful, the app had to provide the following functionality:

- Use the microphone to record sound samples and calculate the decibel value.
- Display the current, average, minimum, and maximum decibel value textually on the screen.
- Display the current decibel value graphically using a line chart.
- Add geolocalisation information to each data sample.
- Draw each data sample on the map using appropriate colour coding.
- Upload each data sample to the observatory.

There were basically two different ways in which these features could be implemented: either through the use of a few high-level composite components, or by manually recreating this component composition using lower level components. The students were given a total of 30 minutes to accomplish this goal. Part of this time was spent on navigating the meta-platform's website, creating a new account for each student, creating an observatory, etc. At the end of the experiment, the students were asked to fill in a survey to evaluate DISCOPAR^{DE} and the CO meta-platform as a whole.

Experiment Results

None of the students experienced difficulties with the meta-platform's website navigation and they all managed to create an account and a novel citizen observatory¹. After 12 minutes, the first student completed the task and created a functioning mobile app. In the end, 8 out of 9 students designed an app that satisfied the intended functionality requirements within the time limit. One student forgot to ensure that each data sample needs to be uploaded to the observatory's server.

¹Some students lost some time downloading a different browser for compatibility reasons.

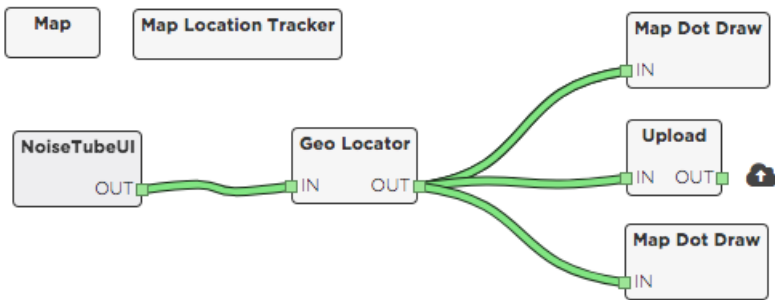


Figure 7.14: Noise measuring app implementation with redundancies.

Despite the fact that 8 out of 9 students managed to implement a functioning mobile app providing the necessary features, 5 solutions included redundant or even unused parts in their graph. Consider for example the implementation depicted in fig. 7.14. This student’s solution utilises the high-level `NoiseTubeUI` which already includes the required visualisations to display current decibel values and additional statistics. Each observation produced by this component is correctly tagged with GPS coordinates using `GeoLocator`. However, at this point, each observation is sent to two instances of the `MapDotDraw` component. Having multiple instances of this component (connected to the same output port) serves no purpose, as they both do exactly the same thing, namely drawing an observation on the map. Therefore, each observation is always drawn twice on the app’s map. Similar redundancies were implemented by two other students. For example, one student uses both a `MapDotDraw` and `MapMarkerdraw` component, meaning that the location of every observation was also indicated using a plain marker on the map.

Two other students made the mistake of leaving disconnected components on the canvas. These sub-graphs contained processes expecting an input which was not provided due to the lack of an incoming connection. As such, the entire sub-graph will never execute. Although these have no impact on the correct functioning of the required features, they unnecessarily consume memory and/or clog up the UI.

In order to prevent users from implementing such redundancies, we aim to improve `DISCOPARDE`’s graph validation mechanism (cfr. section 4.3.4).

Three students implemented the app as requested without any redundant components. One of these students even opted for the more difficult, low-level, implementation. This student’s implementation is depicted in fig. 7.15. This low-level approach requires the use of several low-level components, some of which are not visible unless the advanced mode is toggled in the component menu.

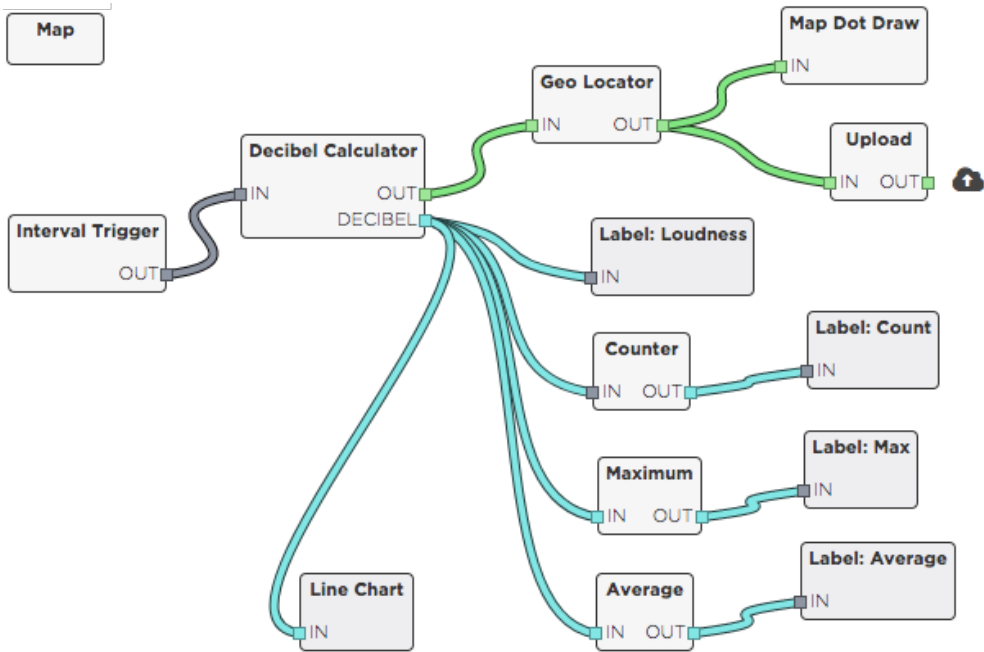


Figure 7.15: Correct, low-level noise measuring app implementation.

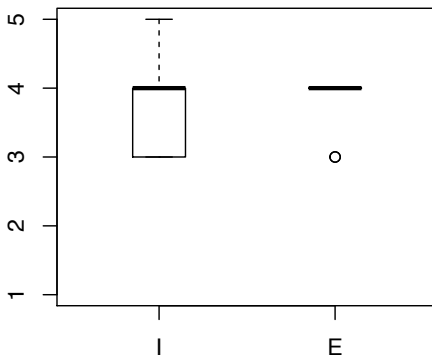
Navigating DISCOPAR^{DE} seemed straightforward and intuitive for most students, as they realised that each component can be further configured through its corresponding configuration window (cfr. section 4.3.3). For example, in the low-level implementation, the `Interval Trigger` is configured to trigger the recording of a sound sample at frequent intervals, each `Label` was modified to display the appropriate text in the UI, and the `Linechart`'s size was updated to take up the remaining space of the UI.

Survey Results

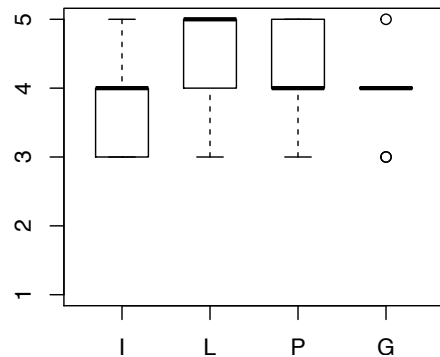
After the quasi-experiment, students were asked to fill in a survey. The survey contained two parts: the first focussed on the participants' experience using DISCOPAR^{DE}, while the latter questioned their interest in the concept of a citizen observatory meta-platform as a whole. An exhaustive list of the survey questions used is available in appendix B.

From fig. 7.16a we can conclude that the drag-and-drop interface of DISCOPAR^{DE} was perceived as intuitive and easy to use by the participants. This is further confirmed by the fact that 8 out of 9 student managed — albeit including

some redundancies — to compose a functioning mobile app that had the envisioned features. Every student soon found out that connections can be made through drag-and-drop interactions. 3 students did not realise that components can be deleted (by right-clicking them), which explained the unused components left on the canvas by 2 of these students. The third student simply created a new citizen observatory to start from scratch.



(a) Boxplot on Intuitiveness (I) and ease of use (E) of the visual programming environment of DISCOPAR.



(b) Boxplot of participants' appreciation of the features of DISCOPAR^{DE}. (I) Component Information (L) Live Preview (P) Port Typing (G) Graph Validation.

Figure 7.16: Survey results (part 1).

All students noticed that ports have different colours, and that certain components cannot be interconnected (cfr. section 4.3.2). However, only 8 out of 9 students noticed the port highlighting mechanism based on matching output ports and input ports, and from those 8 only 4 students understood the meaning of this mechanism and made the link with the compatibility of components. A suggestion was made to add a notification, explaining the highlighting mechanism, that triggers whenever the graph designer attempts to make an illegal connection. Additionally, a large interest was shown in a legend explaining the data type of each port colour.

Every student realised that components expect input from ports on the left side of the visual representation, and produce output on the right side. However, only 4 out of 9 students understood that a component will never execute if it has an input port without any incoming connections. They suggested to add an unused input port as a warning to the graph validation mechanism.

Only 2 students used the search function of DISCOPAR^{DE} and successfully found the component they were looking for. We conclude that the relative low number of

components that is currently present in the component library does not warrant the use of the search function yet.

7 students noticed the information button of components in the component menu, which shows the component's description when clicked. This additional information helped 5 students to understand the functionality of a certain component they did not know before.

The graph validation mechanism (cfr. section 4.3.4) was discovered by 6 students, who each understood its meaning and the errors it listed. However, due to the number of participants that included unused components or duplicate instances in their implementation, we have to conclude that additional checks must be implemented to provide more feedback to the graph designer about the quality of their implementation.

Participants also gave their opinion about the component information, live preview, port typing, and graph validation mechanisms present in DISCOPAR^{DE} by assigning a rating between 1 to 5 for each of these categories. At this point in the quasi-experiment, participants received an explanation of each mechanism, as we noticed from the earlier questions that not every participants discovered or understood the mechanism. The summary of participant opinions is depicted in fig. 7.16b. The most appreciated feature of the Mobile App Design Interface is the live preview that immediately reflects changes made to the implementation of the mobile app. Students reported that this feature greatly increased their understanding of their actions and the implementation of the mobile app as a whole.

In the second part of the survey, students answered questions regarding the very notion of a CO meta-platform. Figure 7.17a contains the results of these questions. The first column contains the answers to the question of how likely they themselves would use the platform for designing environmental data collection apps in future research. The second question focussed on data analysis and visualisation, and asked how likely they were to use a visual interface, such as the Campaign Analysis Interface, as an alternative to existing software such as Microsoft Excel, SPSS, and MatLab, taking into account that DISCOPAR would provide the same functionality. The last question asked their opinion about the usefulness of a full-fledged CO meta-platform for environmental scientists in general.

The group of students that participated in this experiment was the same group that was used for the noise mapping campaign discussed in section 7.2.1. We therefore also took the opportunity to test whether or not their experience with the CO meta-platform and data gathering for the campaigns had any impact on their view of participatory sensing in general. Figure 7.17b indicates the students' general confidence

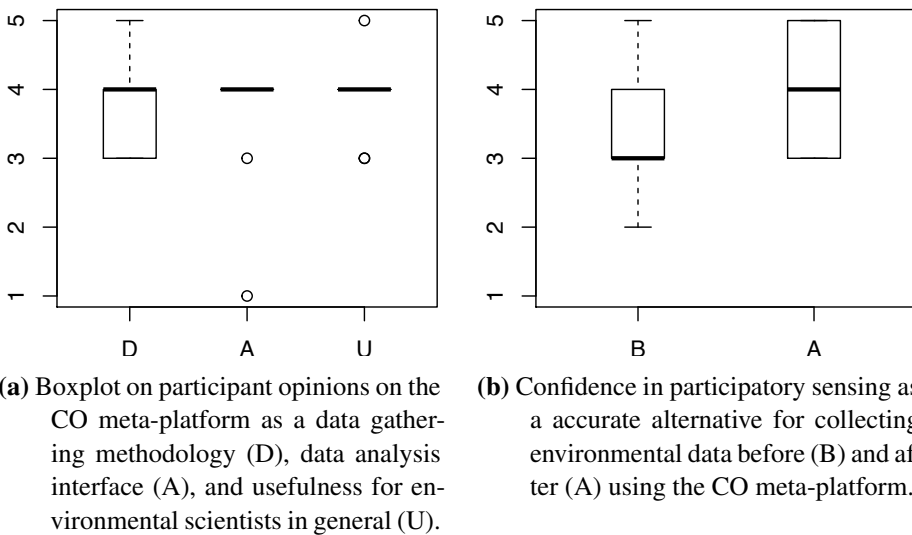


Figure 7.17: Survey results (part 2).

in participatory sensing as an accurate alternative for collecting environmental data before (B) using the CO meta-platform and after (A) interacting with the platform and participating in the campaigns. 5 out of 9 students indicated that — as a direct result of their participation — they now also consider participatory sensing as a valid alternative for collecting scientific data.

7.3.2 Quasi-Experiment: Campaign Design Interface

A second quasi-experiment was performed during the “Reclaiming the city” workshop organised by Redelijk Eigenzinnig, an interdisciplinary elective for students at the Vrije Universiteit Brussel. The course’s aim is to stimulate students to reflect critically on important social issues. The topic of this particular workshop was “accelerated technological development in the field of ICT, app, and sensing that are transforming our cities”.

The idea for this workshop was to give students the opportunity to design their own environmental noise mapping campaign in an area that was in the direct proximity of the workshop’s location, and to collect data with their smartphone using the NoiseTube^{2.0} mobile app. 10 students participated in the workshop, including students from bachelor and master programs in History, Industrial Science, Political Science, Philosophy, etc. Unlike the students from the first experiment, students participating in this workshop received a presentation explaining the concept of our CO

meta-platform and a very basic introduction on DISCOPAR^{DE}. Next, the students were given 30 minutes to program their own campaign and an additional 30 minutes to go out in the field and collect data for their newly created campaigns. fig. 7.18 depicts the output of one such campaign entirely created by one of the students. This campaign uses a P_A that filters out measurements made outside the vicinity of 3 street segments, and aggregates the remaining measurements in a map.

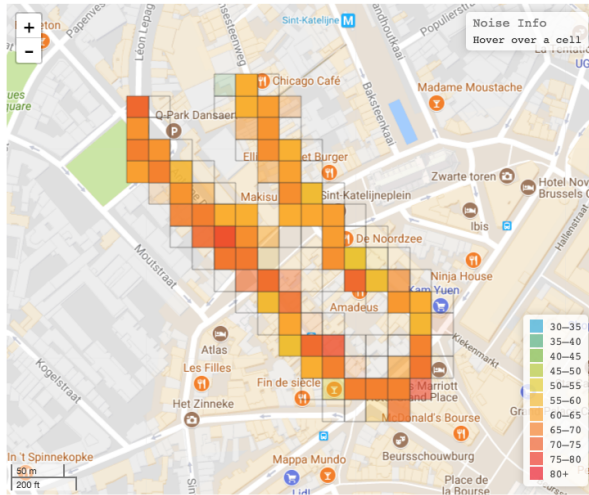


Figure 7.18: Campaign created during the “Reclaiming the city” workshop.

Each participant successfully created a campaign within the foreseen time slot. Due to time restrictions, students were not asked to fill in a survey about their experience with the Campaign Design Interface. However, the fact that all the students successfully created a campaign and managed to collect data for their campaign serves as anecdotal evidence that the Campaign Design Interface is indeed usable by ICT-agnostic users.

7.4 Conclusion

In this chapter, we demonstrated how the CO meta-platform developed in this dissertation lives up to our research vision which consists of building a generic approach towards reusable and reconfigurable citizen observatories that is accessible by ICT-agnostic users.

Through the creation of three radically different citizen observatories, we demonstrate that our citizen observatory meta-platform provides support for both continuous

data streams (e.g., sensorial data such as sound pressure levels captured through a microphone) as well as discrete data (e.g., such as pedestrians' reports on the quality of informal walking trails). Furthermore, mobile app created through the meta-platform can be seamlessly augmented with external sensors as long as they can connect to the mobile phone over a Bluetooth connection. As a result, more types of data can be gathered than the ones that are directly available through a smartphone's built-in sensors. This wide-range support of data types means that our CO meta-platform can be considered, to the best of our knowledge, the first of a kind in new and more powerful generation of reusable participatory sensing systems such as those presented in section 2.2.2.

The CO meta-platform further distinguishes itself from existing PS systems through its integrated technological support for participatory campaigning. Campaigns were already performed in the past with several existing PS systems, but they always relied on human effort to enact and monitor the campaign and guide participants. In case of the CO meta-platform, the notion of a campaign, the means to design a campaign, and automated campaign orchestration are all part of the meta-platform. Furthermore, campaigns can utilise participant coordination mechanisms to increase data quality by optimising coverage, density, etc.

In section 2.4.3 we discussed how providing real-time feedback to participants has multiple benefits. Feedback can be used to provide participant coordination on an individual level, but also acts as an incentive for participants when they see the effect of their contributions. Unlike other PS systems that rely on a more static approach using batch-processing and queries, citizen observatories and campaign created on our CO meta-platform process and analyse data in real-time due to the reactive nature of DISCOPAR. This reactive approach enables real-time (individual) participants feedback and coordination. Furthermore, data visualisations on an observatory or campaign's dashboard automatically stay up to date.

Perhaps the biggest ground-breaking feature of our CO meta-platform is that it enables people with limited technological knowledge to create their own citizen observatory and set up PS campaigns. We observed from the two different quasi-experiments discussed in section 7.3 that non-ICT experts can indeed implement their own mobile data gathering app and set up a campaign with ease. These quasi-experiments only focussed on the Mobile App Design Interface and the Campaign Design Interface, so further and more elaborate user studies are necessary to evaluate the CO meta-platform as a whole. However, we assume that the other interfaces provided by the CO meta-platform, such as the observatory's Data Processing Design Interface, will be equally well-received and accessible to ICT-agnostic users as they are all built on top of DISCOPAR^{DE}.

8

CONCLUSION

This dissertation introduces the idea of a citizen observatory meta-platform, i.e., a platform that enables the construction of citizen observatory platforms. This final chapter summarises the functionalities of our CO meta-platform and the visual programming language that it offers, provides an overview of the contributions of this dissertation, and highlights some directions for future work.

8.1 Summary

The CO meta-platform presented in this dissertation enables ICT-agnostic users to construct their own citizen observatories. This is achieved through the use of DISCOPAR, a new visual flow-based domain-specific programming language designed to hide the non-essential complexity of constructing citizen observatories from the end-user, and to only confront them with concepts that are truly relevant to their domain. The advantages of our CO meta-platform over other existing PS systems are its reconfigurability, its integrated support for automatically enacting campaigns, reactivity, and its usability by ICT-agnostic users (cfr. section 2.2.1 and section 2.2.2).

Reconfigurability The CO meta-platform enables the creation and configuration of citizen observatories for a large spectrum of PS scenarios. Citizen observatories are capable of collecting different types of data, including both sensorial parameters (noise, accelerometer, humidity, etc.) and behavioural parameters (e.g., user input through questionnaires). In addition to providing the necessary features to collect this diverse data, the CO meta-platform also enables the configuration of different data processing algorithms for each citizen observatory scenario. The CO meta-platform is capable of offering this high degree of reconfigurability thanks to the flow-based nature of DISCOPAR, which enables a wide variety of applications to be built from the same set of components through different compositions. Thanks to the component-based nature of FBP, the CO meta-platform can avoid rebuilding all the resulting mobile apps, web interfaces, databases, data analysis and visualisation elements from scratch.

Campaign Support Unlike existing PS systems where the concept of a campaign is not explicitly supported and thus usually managed manually, the CO meta-platform provides integrated support for the definition of a PS campaign as well as for its automated enactment. Within each observatory created on the meta-platform, initiators can define campaigns themselves. Campaigns provide automated orchestration of participants while they are collecting data and provides them with immediate feedback on their contributions as an incentive mechanism.

Reactivity Each citizen observatory created using the CO meta-platform is a cloud-based reactive application that reacts to data coming from mobile devices that contribute the data to the server, and that promptly pushes feedback (e.g., intermediate campaign results) back to the relevant devices. This reactivity enables real-time automated orchestration to guide data collection and provide immediate user feedback,

essentials tools for successful campaigning. This also means that, unlike existing PS systems, every form of data visualisation is always kept up to date and does not rely on batch-processing techniques a posteriori.

ICT-Agnostic Usability DISCOPAR^{DE}, the visual development environment for DISCOPAR, provides ICT-agnostic users the means to construct a citizen observatory by visually composing components through drag-and-drop actions. DISCOPAR^{DE} greatly increases the usability of DISCOPAR by only presenting the graph designer with the essential domain-specific concepts, thereby hiding the irrelevant technological difficulties that normally come with deploying a cloud-based reactive application.

8.1.1 Contributions

This section gives an overview of the contributions of this dissertation.

Participatory Sensing Systems: State of the Art We gave an overview of the history and evolution of participatory sensing, where a distinction is made between ad-hoc PS systems that are designed on a per use-case basis, and reusable PS systems. Until now, the latter only focus only on discrete data, i.e., single-shot observations usually collected through digital questionnaires (cfr. section 2.2). We define the concept of a PS campaign (cfr. section 2.3.2) and identify the typical predicates used in the definition of a campaign’s protocol (cfr. section 2.3.1). Furthermore, from analysing several environmental PS campaigns (combined with our extensive expertise in participatory noise sensing gained from the NoiseTube project [137, 41]), we derive and describe each step of a typical campaign’s lifecycle (cfr. section 2.3.3). We also introduce the concept of a citizen observatory (cfr. section 2.4).

Citizen Observatory Meta-Platform We have coined the term citizen observatory meta-platform, which is a platform to construct citizen observatory platforms. The CO meta-platform presented in this dissertation (cfr. chapter 5) is a contribution in the field of participatory sensing, as it can be considered the first of its kind in a new generation of reusable and reconfigurable PS systems:

- The CO meta-platform provides support for a wide range of data types that can be collected by the apps it allows us to construct (including both sensorial parameters and behavioural parameters).
- The CO meta-platform features integrated technological support for the definition and automated enactment of campaigns.

- The CO meta-platform further distinguishes itself from existing work due to its reactive nature, which enables real-time campaign orchestration and feedback.
- The CO meta-platform remains accessible for ICT-agnostic users allowing anyone to create their own citizen observatory and deploy campaigns.

DISCOPAR Citizen observatories created through the CO meta-platform are collaborative cloud-based reactive applications. In order to support real-time communication amongst the various components deployed in a citizen observatory's distributed architecture, we developed DISCOPAR, a new visual reactive flow-based domain-specific language, created specifically to hide the non-essential complexity of citizen observatories and their distributed nature from the end-user, and to present only concepts that are relevant to their domain (cfr. chapter 4). To the best of our knowledge, this is the only flow-based programming language where components are automatically distributed amongst devices and their corresponding connections automatically established.

DISCOPAR^{DE} In order to enable ICT-agnostic users to create their own citizen observatories and define their own campaigns, we have developed DISCOPAR^{DE}, a web-based visual programming environment for DISCOPAR (cfr. section 4.3). It features a live programming mode that allows the direct manipulation of a running DISCOPAR program without stopping it (cfr. section 4.3.5). Moreover, DISCOPAR^{DE} enables users to perform distributed live programming, which means that they can change the observatory's server-side logic from within a web-browser (i.e., remotely). To help ICT-agnostic users create correct programs in DISCOPAR, DISCOPAR^{DE} features a graph validation mechanism that prevents them from composing incompatible components.

Citizen Observatory Meta-Platform Validation We have validated the citizen observatory meta-platform in terms of expressiveness, suitability, and usability through experiments in both laboratory as well as real-world conditions. The expressiveness and suitability of DISCOPAR for constructing citizen observatories is demonstrated through the creation of three distinct citizen observatories and the successful enactment of a number of campaigns designed within those observatories (cfr section 7.2). Usability of the CO meta-platform is validated through two end-user usability tests (quasi-experiments) where we provide some evidence that people without any programming knowledge can successfully create a citizen observatory and design a campaign using the CO meta-platform (cfr. section 7.3). From the analysis of a partici-

pant's survey, we analysed the participants' experience using DISCOPAR^{DE} and their interest in the concept of a citizen observatory meta-platform as a whole. Both were well-received.

8.2 Future Work

This section outlines ongoing work and provides several possible avenues for future research. First, we discuss the advantages of shifting the implementation of mobile apps from pure web-based apps to hybrid mobile apps. Second, we focus on optimising performance by enabling components to run on different threads. Last, we present various ideas towards further enhancing the CO meta-platform with regards to user experience.

8.2.1 Hybrid Mobile Apps

In the current status, mobile apps created through the CO meta-platform are implemented as web-based HTML5 applications (cfr. section 6.3.1). Although web-based apps are becoming increasingly powerful, new APIs only become mainstream very slowly and therefore often lack support by all browser vendors. Furthermore, there are still some limitations to hardware access, and offline availability is more difficult to implement and is also not without limitations. In ongoing work, we are switching to a hybrid approach by using PhoneGap [59] to facilitate and enhance the implementation of offline availability and hardware access.

Offline Support

Offline support can be implemented in web-based apps using to JavaScript's Service Workers. However, there are some limitations and inconsistencies involved that make them not yet a recommended technology to implement offline-capable mobile data collection apps. In case of citizen observatories, participants can potentially collect huge amounts of data using their smartphone, especially when sensorial data is being collected. When operating offline, all these data have to be persistently stored until an internet connection is available. Web-based apps can therefore either rely on Web Storage (e.g., `LocalStorage` and `SessionStorage`) but this is limited in size and type (strings only). Alternatively, apps can utilise the File System API, although this API is also not sufficiently mature or standardised to encourage widespread adoption yet. In contrast, hybrid mobile apps provide the benefits of a native app, i.e., access to the smartphone's internal storage. Therefore, hybrid apps provide a more straightforward

and mature approach when it comes to persistently storing data on a phone. Having no restrictions on the type of data that can be stored is also an important requirement, as certain observatories may need to store photos, videos, etc. In the future, we will focus on increasing the mobile app's user experience and on making it more robust, thereby particularly focussing on offline availability.

Access to Hardware

In addition to offline-capabilities, another advantage of hybrid mobile app technology is that hybrid apps do not have any restrictions regarding access to the mobile device's hardware. This is advantageous for both sensor access and communication with external devices through Bluetooth and USB.

In section 2.1, we discussed how smartphones are becoming increasingly powerful and equipped with multiple sensors. Currently, many sensors (i.e., GPS, accelerometer, microphone, ambient-light, camera, and magnetometer) are only accessible through (experimental) APIs of *some* web browsers, as they are often still considered non-standard APIs. A hybrid approach solves this issue by providing direct access to the sensors rather than going through a web browser's API. Additionally, a hybrid app provides access to more uncommon sensors, such as the on-board temperature and humidity sensors that some smartphone models provide.

Bluetooth communication is another technology that is not well-supported by web browsers. At the time of writing, only Chrome supports the Web Bluetooth API. Furthermore, it requires an external device featuring the Bluetooth 4.0 standard, which introduced a new "Low Energy" mode and provides most of its functionality through key/value pairs provided by the Generic Attribute Profile (GATT). Hybrid mobile apps do not have such limitations and are thus capable of communicating with a wider range of external devices, including any version of Bluetooth. Furthermore, having access to the device's hardware also enables serial communication through USB.

In ongoing work, we successfully made two additional citizen observatories: one focused on gathering particulate matter data by communicating with an external sensor through Bluetooth 2.0, and another one collecting black carbon data by establishing a USB serial connection to an external device.

8.2.2 Performance

In the current ongoing work, every process (cfr. section 4.2.2) runs within the same thread due to Javascript's single-threaded execution model. Although this setup works fine for the relatively small-scale experiments described in chapter 7, it is not

capable of handling “big data” scenarios, where thousands of participants are all simultaneously contributing data to an observatory. To prevent bottlenecks and ensure real-time feedback of the observatory, it is important that the server-side processing graphs can deal with the increased amount of data. Therefore, we are investigating possible methods to increase the scalability of observatories, such as multi-threading and process replication.

Multi-threading

The first avenue to increase performance is to implement a thread-pool that automatically manages where processes are deployed based on the current workload of each process. We envision various predicates that can be placed upon a process to decide when it should be deployed in a separate thread (e.g., based on a certain throughput indicator). When redeploying a process, only the connections from and to that process have to be updated, since processes only communicate with their input and output ports and are not concerned with the connections attached to their ports. This means that processes do not know the difference whether they are communicating to a process in the same thread or a different one. The amount of data that a citizen observatory must process in real-time varies greatly as the number of participants actively contributing data to it can differ significantly throughout the observatory’s lifetime. Certain processes can therefore suddenly be required to process a lot more data. Therefore, load balancing (i.e., assigning a process to a different thread at runtime) should also be supported.

Process Replication

The second avenue is to implement a replication mechanism at the level of processes. Rather than replicating an entire observatory, the flow-based programming approach of DISCOPAR enables citizen observatories to only scale processes that are causing a bottleneck. For example, if two interconnected processes produce/consume data at a different rate (e.g., the first process produces data faster than the second can consume), there is a risk of buffer overflow. The goal is to automatically detect these situations and create multiple instances of the “slower” process. There is an additional complexity to take into account for non-functional processes. Processes that maintain state, such as an aggregated map, need to operate on the same shared object and thus have a shared state.

8.2.3 User Experience Enhancements

DISCOPAR^{DE} is used in our CO meta-platform to enable ICT-agnostic users to set up a citizen observatory and deploy campaigns within a citizen observatory. Although the CO meta-platform is fully functional at the time of writing, there are several potential improvements that can be applied with the goal of having a positive effect on the user experience. We discuss some of these in the remainder of this section.

Graph Auto-Complete

In section 4.3.4 a validation mechanism was introduced that verifies whether certain constraints imposed on the graph are satisfied. In the current state, this mechanism is rather basic and only supports three types of constraints (cfr. section 6.1.5). In the future, the types of constraints supported will be increased. However, the same graph validation mechanism could also be used to automatically make suggestions to the graph designer, enabling a form of “auto-complete” mechanism. For example, when the graph validation mechanism notices that a certain process requires input but has no incoming connections yet on that input port, it could suggest to create a connection from an unused output port from a process placed nearby on the canvas.

WYSIWYG Editor

Currently, every observatory’s mobile app uses a default layout (cfr. section 5.1.3). Although the active components of each observatory’s mobile app can be programmed using DISCOPAR^{DE}, there is no means to customise the app’s UI. For example, it is not possible to include a custom logo, or use a different colour scheme. Although it is relative easy to modify the default app look (e.g., as demonstrated by the Trage Wegen app shown in section 7.2.2), it is currently only possibly by additional coding in HTML5 and CSS. A better alternative would be to include a more advanced WYSIWYG editor that enables full customisation of the mobile app’s layout. This editor would enable us to modify every visualisation component, in addition to providing basic layout elements such as images and different types of navigational menus. Whenever such a visualisation component is included in the layout, a corresponding process should be added to the graph, enabling the graph editor to connect the visualisation to other processes.

Additionally, a WYSIWYG editor could be also be used to customise the appearance of the various web pages that are automatically created by the CO meta-platform for a new citizen observatory and campaigns.

Process Groups and Zoom Controls

DISCOPAR^{DE} is faced with a problem that all visual programming languages have in common: scalability. Applications that are composed of a very large number of components can become so complex that interpreting their visual representation becomes very difficult or even impossible. This can be solved by varying the granularity of the program shown at a given moment. By grouping components hierarchically, they can be reduced into a single component representing that group. Similar to how community components are implemented (cfr. section 5.3.2), this grouped component would expose input ports and output ports for connections going to processes outside the group. At any time a group can be expanded, enabling the user to alter its containing components.

The user-friendliness can also be further increased by adding zoom controls and better mechanics to navigate the canvas of the visual programming environment.

Graph Templates

Newly created citizen observatories and campaigns use a default graph template for their various interfaces. For example, as discussed in section 6.3.2, a new citizen observatory's Data Processing Design Interface (cfr. section 5.1.2) starts with a default graph template containing some basic functionality, such as storing data in the database.

In the future, we would like to use this graph template mechanism as a means for users of the CO meta-platform to recycle the "code", i.e., the component composition, of an already existing citizen observatory or campaign. For example, consider a user inspired by an existing campaign who wants to design a similar campaign. In the current status, this user has to manually re-create this campaign and modify it where needed. It would be more efficient if that user could somehow copy-paste the existing campaign's implementation and start by modifying the copy. Therefore, we would like to enhance the graph templates, so that existing citizen observatories and campaigns can be reproduced and modified with ease. Furthermore, these graph templates could also be used to have a series of demonstrative showcases of the platform's features. We envision various mobile app graph templates where users can select one that most closely resembles their goal.

8.3 Closing Remarks

Despite the evolution of participatory sensing as a data collection paradigm and the emergence of several citizen observatories, there is a lack of reusable and reconfigurable citizen observatory construction tools. Although there already exist a couple of reusable and reconfigurable PS systems, these primarily focus on the data gathering (typically form-based) thereby neglecting the other roles of a citizen observatory such as advanced data processing methods, support for campaigns, and real-time participant feedback and coordination. This led us to the idea of a citizen observatory meta-platform. A CO meta-platform offers a generic approach towards reusable and reconfigurable citizen observatories. Through this CO meta-platform, ICT-agnostic users can construct their own citizen observatories and design PS campaigns autonomously. The CO meta-platform provides support for a wide range of data types, including both sensorial parameters and behavioural parameters, and also features integrated support for the definition and enactment of campaigns. These features enable a new generation of PS scenarios where we move away from small-scale research-oriented deployments to the full-fledged adoption of PS as a societally and scientifically relevant method.

To conceive the first CO meta-platform, we designed DISCOPAR, a new visual reactive flow-based domain-specific language, which automatically handles the distributed nature of citizen observatories and that only presents essential domain-specific concepts to users. The flow-based nature of DISCOPAR means that the entire system is reactive and capable of processing data and providing feedback in real-time. To ensure the usability of the DISCOPAR language by ICT-agnostic users, we developed DISCOPAR^{DE}, a web-based visual programming environment for DISCOPAR. Throughout the CO meta-platform, DISCOPAR^{DE} is used to enable ICT-agnostic users to design the various parts of a citizen observatory. We demonstrated the expressiveness and suitability of the CO meta-platform through the creation of different citizen observatories and the successful enactment of campaigns deployed within these observatories. We have shown the ICT-agnostic usability of the CO meta-platform through usability experiments involving people who lack any programming knowledge.

We are confident that DISCOPAR and DISCOPAR^{DE} merely scratch the surface of a promising new strand of languages and systems that will form the technological basis for truly democratising the citizen science approach. Our technologies not only allow citizens to participate in citizen science experiments that were set up by experts, but also empower them to set up their own experiments and invite their fellow citizens to participate in those experiments in an open and democratic fashion.

A

COMPONENT LIST OF DISCOPAR

A.1 Sensing Components

COMPONENT NAME	INPUT PORT(S)	OUTPUT PORT(S)	DESCRIPTION	SETTINGS
SensorDrone		OUT (Observation) TEMPERATURE (Numeric) HUMIDITY (Numeric) PRESSURE (Numeric) BATTERY (Numeric)	Connects to the SensorDrone Bluetooth sensor and outputs its temperature, humidity, and pressure readings.	
SoundPressureLevel	IN (Any)	OUT (Numeric)	Records a sound sample using the microphone whenever the IN port receives data and calculates the sound pressure level thereof.	
SoundLevelMeter	IN (Any)	OUT (Observation) DECIBEL (Numeric)	Records sound samples using the microphone and calculates the dB(A) value of these samples.	
NoiseTubeUI		OUT (Observation)	Records sound samples using the microphone and calculates the dB(A) value of these samples and return it as data token. The dB(A) is also automatically displayed and added to a graph.	nrOfSamples (15) yAxisMax (100) yAxisMin (0) yAxisInterval (10) chartHeight (100%) colourScale (Black)
Acceleration		X (Numeric) Y (Numeric) Z (Numeric)	Output the data from the smartphone's acceleration sensor.	
Form		OUT (Observation)	Enables you to construct a form using a drag-and-drop interface. The form will be added to the mobile app to enable survey-based data collection	
Camera		OUT (Observation)	Take photos using the camera. Alternatively, use the "Form" component to create a questionnaire which can include photos as well	
GeoLocator	IN (Observations)	OUT (Observation)	Adds GPS information to observations.	
WatchGeoLocation		OUT (Location)	Sends out a location each time the user's location changes.	
SpeedCalculator	IN (Observation)	OUT (Observation) SPEED (Numeric)	Calculates the speed based on two consecutive observations with GPS locations attached to them.	
Speed		OUT (Observation) SPEED (Numeric)	Outputs the speed a mobile app user is moving at.	

A.2 Logic Components

A.2.1 Geographical Components

COMPONENT NAME	INPUT PORT(S)	OUTPUT PORT(S)	DESCRIPTION	SETTINGS
IsInsidePolygon	IN (Observation)	OUT (Observation) FALSE (Observation)		
IsInsideMultiPolygon	IN (Observation)	OUT (Observation) FALSE (Observation)		
IsNearLine	IN (Observation)	OUT (Observation) FALSE (Observation)		distance
IsInsideCircle	IN (Observation)	OUT (Observation) FALSE (Observation)		distance
GeographicalFilter	IN (Observation)	OUT (Observation) FALSE (Observation)	Filters out observations that are not made within the specified geographical area.	

A.2.2 Temporal Components

COMPONENT NAME	INPUT PORT(S)	OUTPUT PORT(S)	DESCRIPTION	SETTINGS
TimeFilter	IN (Observation)	OUT (Observation)	Filter based on a time window: Only observations made inside the specified time window are forwarded.	from (12:00:00) to (13:00:00)
DateFilter	IN (Observation)	OUT (Observation)	Filter based on a date window: Only observations made inside the specified date window are forwarded.	from (2018-06-01) to (2018-06-31)
DayFilter	IN (Observation)	OUT (Observation)	Filter based on day of the week: Only data tokens made on the specified day of the week are forwarded.	day (Monday)
DayFilterSplit	IN (Observation)	MONDAY (Observation) TUESDAY (Observation) WEDNESDAY (Observation) THURSDAY (Observation) FRIDAY (Observation) SATURDAY (Observation) SUNDAY (Observation)	Filter based on day of the week and outputs observations on the matching output port.	

A.2.3 Contextual Components

COMPONENT NAME	INPUT PORT(S)	OUTPUT PORT(S)	DESCRIPTION	SETTINGS
FilterCampaignParticipants	IN (Observation)	OUT (Observation)	Only allows observations made by participants of the campaign to pass.	

Appendix A: Component List of DISCOPAR

A.2.4 Miscellaneous

COMPONENT NAME	INPUT PORT(S)	OUTPUT PORT(S)	DESCRIPTION	SETTINGS
SnapToRoad	IN (Location)	OUT (Location)	Snaps a coordinate to the street network.	
IntervalTrigger		OUT (Any)	Outputs a signal on a configurable interval that can be used to trigger other components.	interval (1000ms)
Console	IN (Any)		Prints received information packets to the console. Only used for debugging purposes.	
Counter	IN (Any)	OUT (Numeric)	Counts the number of information packets received and outputs the counter's value.	
Maximum	IN (Numeric)	OUT (Numeric)	Keeps track of the maximum value provided as input, and outputs the new maximum whenever it changes.	
Minimum	IN (Numeric)	OUT (Numeric)	Keeps track of the minimum value provided as input, and outputs the new minimum whenever it changes.	
Average	IN (Numeric)	OUT (Numeric)	Keeps track of the average value provided as input, and outputs the new average whenever it changes.	
Filter	IN (Any)	OUT (Any)	Filters input based on a specified predicate.	predicate
DataCollectionQuery		OUT (Observation)	Outputs every observation in a data collection once when initialising the component.	
Database		OUT (Observation)	Outputs every observation in the citizen observatory's database.	
ObservationMaker	IN (Numeric)	OUT (Observation)	Creates an observation from raw data values	
ObservationData	IN (Observation)	[1 output port for each value in the observation] (depends on observation data content)	Outputs the raw data values contained within an observation.	
FilterFormData	IN (Observation)	(depends on the type of form field)	Filters observations based on the value of a specific field of the form. These components are only available when the mobile app uses a form to gather data from the participants.	
FilterRadioField	IN (Observation)	[1 output port for each option in the radio field] (Observation)	Filters observations based on the value of a specific radio field of the form. These components are only available when the mobile app uses a form to gather data from the participants.	
FilterCheckBoxField	IN (Observation)	[1 output port for each option in the checkbox field] (observation)	Filters observations based on the value of a specific checkbox field of the form. These components are only available when the mobile app uses a form to gather data from the participants.	
FilterTextField	IN (Observation)	TRUE (Observation) FALSE (Observation)	Filters observations based on the value of a specific text field. These components are only available when the mobile app uses a form to gather data from the participants.	operator compareTo
CalibrateDecibel	IN (Observation)	OUT (Observation) DECIBEL (Numeric)	Calibrates decibel values of an observations using a calibration profile.	
Upload	IN (Observation)	OUT (Observation)	Uploads observations to the observatory	
ApplicationData	IN (Observation)	OUT (Observation)	Receives observations uploaded by the mobile app	

A.3 Aggregation components

COMPONENT NAME	INPUT PORT(S)	OUTPUT PORT(S)	DESCRIPTION	SETTINGS
AggregatedMap	IN (Observation)	OUT (GeoJSON)	Creates an aggregated map of the specified area. Output: aggregated map (geojson) that can be send to visualisation components	field
Store	IN (Observation)	OUT (Observation)	Persistently stores observations in the citizen observatory's database.	
DataCollection	IN (Observation)	OUT (Observation)	Creates a persistant data collection of all the data samples it receives.	name
MapDataCollection	IN (GeoJSON)		Persistently stores maps in GeoJSON format	name

A.4 Coordination components

COMPONENT NAME	INPUT PORT(S)	OUTPUT PORT(S)	DESCRIPTION	SETTINGS
PushNotification	IN (Message)		Triggers a push notification on the mobile app upon receiving a message.	
ArrowIndicator	FROM (Location) TO (Location)		This components draws an arrow on the map to guide the user to a given location.'	
ParticipantMovement		OUT (Observation)	Outputs an observation containing a location whenever the mobile app user's location changes. Used by coordination algorithms.	
ProximityNotification	IN (Observation)	OUT (Message)	Will inform a participant whenever s/he is within the specified distance of a campaign's objective.	message title distance (10m)
FindNearestObjective	OBSERVATION (Observation) OBJECTIVE (GeoJSON)	OUT (Message)	This component instructs people to go the nearest campaign objective that still needs data.	
FindNearestGridCell	OBSERVATION (Observation) MAP (GeoJSON)	OUT (Message)	This component instructs people to go the nearest grid cell that still needs data.	

A.5 Visualisation components

COMPONENT NAME	INPUT PORT(S)	OUTPUT PORT(S)	DESCRIPTION	SETTINGS
Label	IN (Any)		This component adds a label to the mobile app, that displays the most recent value of the connected input.	heading unit
LineChart	IN (Numeric)		This component expects a numeric value, and will display this value on a dynamic chart.	nrOfSamples (15) yAxisMax (100) yAxisMin (0) yAxisInterval (10) chartHeight (100%) colourScale (Black)
Map			This component displays a map and enables you to configure the map layer.	layer (mapBox)
MapLocationTracker			This component centers the map on the current location of the user	
MapDotDraw	IN (Observation)		This component can draw coloured dots on the map to visualize your data using colour coding.	colourScale
MapMarkerDraw	IN (Observation)		This component draws markers on the map for the geo-localised data provided.	
GeoJSONDrawer	GEO (GeoJSON) Zoom (Any)		This component draws any geoJSON data it received on a map.	
TableVisualisation	IN (Observation)		Dashboard visualisation that visualises observations in a table.	height (400px) width (100%) description
AggregatedMapVisualisation	GEO (GeoJSON)		Dashboard visualisation that visualises aggregated maps by drawing their geojson on a map layer	height (400px) width (100%) description
MapMarkerVisualisation	IN (Observation)		Dashboard visualisation that visualises observations samples by plotting them on a map based on their geolocation.	height (400px) width (100%) description
MapVisualisation	IN (Observation)		Dashboard visualisation that visualises observations by plotting them on a map based on their geolocation.	height (400px) width (100%) description
LineChartVisualisation	IN (Numeric)		Dashboard visualisation that visualises data by drawing it on a linechart.	height (400px) width (100%) description nrOfSamples (15) yAxisMax (100) yAxisMin (0) yAxisInterval (10) chartHeight (100%) colourScale (Black)
PieChartVisualisation	IN (Numeric)		Dashboard visualisation that visualises data by creating a pie chart.	height (400px) width (100%) description

B

SURVEY QUESTIONS

Appendix B: Survey Questions

General Information

1. What is your name?
2. What is your age?
3. What is your gender?

DISCOPAR^{DE}

1. On a scale of 1-5, how easy did you find it to work with DISCOPAR^{DE}'s drag & drop interface?
2. On a scale of 1-5, how intuitive did you find it to work with DISCOPAR^{DE}'s drag & drop interface?
3. Did you realise that a component accepts input from the left side, and produces output on the right side?
4. Did you realise that components with no connections arriving at their input port(s) do nothing?
5. Did you notice that certain connections and input/output ports have a different colour?

Y.1 Did you understand their meaning?

6. Did you notice that — while dragging a connection — certain elements become highlighted?

Y.1 What do you think is the meaning of these highlights

7. Did you notice that certain components cannot be interconnected?

Y.1 Did you realise that it was due to the colours of their input and output ports being different?

8. How useful do you think the colour mechanism is to prevent a connection between incompatible components?
9. How useful would you think a legend explaining the difference in colours would be?
10. Did you find out by yourself that you are able to right-click a component to delete / configure it?

11. By right-clicking did you change the configuration of any of the components?
12. Did you use the search function provided by the component menu?
 - N.1 Why did you not use the search function?
 - Y.1 Which keywords did you search for?
 - Y.2 Did the search function help you find the components you were looking for?

Appendix B: Survey Questions

13. Did you click on a component's information button in the component menu?

Y.1 Did the provided information help you to understand the functionality of a component that you did not initially understand?

14. On a scale of 1-5, how useful would you rate the possibility to get more information about a component?

15. Did you click on the "Graph Validation" button when building your mobile app?

Y.1 Did you realize that it verifies whether or not the application you made is valid?

16. On a scale of 1-5, how useful do you think this mechanism of "Graph Validation" is to prevent the creation of a non-functional mobile app?

17. Did you realise the smartphone on the right was a live preview of the application you were building?

18. On a scale of 1-5, how useful do you think this live preview is?

19. In the end, did you manage to build a functional mobile app that recorded sound levels and collected GPS location data?

20. Share any tips, feedback and/or comments you have about DISCOPAR^{DE}.

Citizen Observatory Meta-Platform

1. On a scale 1-5, If given the opportunity, how likely are you to use the platform for environmental data capturing?

2. In case DISCOPAR would provide the same features as programs such as Microsoft Excel, SPSS, Matlab, how likely are you to use DISCOPAR as an alternative?

3. What is your opinion about the following statement: "The visual interface provided by DISCOPAR enables ICT-agnostic scientists to develop Mobile Apps with ease"

4. What is your opinion about the following statement: "Seeing a program as components that can be connected corresponds to how I think about data processing"

5. Taking into account that DISCOPAR is a prototype, how useful do you think the finished platform will be for environmental scientists?

6. On a scale 1-5, BEFORE using the Mobile App (and smartphones as sensor), how confident were you that smartphones could produce accurate data?
7. On a scale 1-5, AFTER using the Mobile App (and smartphones as sensor), how confident are you that smartphones could produce accurate data?
8. Would you say that - as a result of using DISCOPAR - you now also consider smartphones as an alternative approach for gathering scientific data?

BIBLIOGRAPHY

- [1] Amelia Acker, Martin Lukac, and Deborah Estrin. Participatory sensing for community data campaigns: A case study. *Center for Embedded Network Sensing*, 2010. 30
- [2] William B. Ackerman. Data flow languages. *IEEE Computer*, 15(2):15–25, 1982. 44
- [3] Rodina Ahmad. Visual languages: A new way of programming. *Malaysian Journal of Computer Science*, 12(1):76–81, 1999. 73
- [4] Murali Annavaram, Nenad Medvidovic, Urbashi Mitra, Shrikanth Narayanan, Gaurav Sukhatme, Zhaoshi Meng, Shi Qiu, Rohit Kumar, Gautam Thatte, and Donna Spruijt-Metz. Multimodal sensing for pediatric obesity applications. *UrbanSense08*, page 21, 2008. 16
- [5] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978. 44
- [6] Bacon.js. Bacon.js - functional reactive programming library for javascript. <https://baconjs.github.io/>. Accessed: 2018-03-16. 47
- [7] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):52, 2013. 47
- [8] H Bergius. Noflo–flow-based programming for javascript. Accessed: 2017-08-02. 47, 51
- [9] Mark Bilandzic, Michael Banholzer, Deyan Peev, Vesko Georgiev, Florence Balagtas-Fernandez, and Alexander De Luca. Laermometer: A mobile noise

Bibliography

- mapping application. In *Proceedings of the 5th Nordic Conference on Human-computer Interaction: Building Bridges*, NordiCHI '08, pages 415–418, New York, NY, USA, 2008. ACM. 17
- [10] Michael Blackstock and Rodger Lea. Toward a distributed data flow platform for the web of things (distributed node-red). In *Proceedings of the 5th International Workshop on Web of Things*, pages 34–39. ACM, 2014. 58
- [11] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. The reactive manifesto, 2014. 5
- [12] I. Boutsis and V. Kalogeraki. Privacy preservation for participatory sensing data. In *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*, pages 103–113, March 2013. 20
- [13] Waylon Brunette, Mitchell Sundt, Nicola Dell, Rohit Chaudhri, Nathan Breit, and Gaetano Borriello. Open data kit 2.0: Expanding and refining information services for developing regions. In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, HotMobile '13, pages 10:1–10:6, New York, NY, USA, 2013. ACM. 2, 18
- [14] Margaret M Burnett. Visual programming. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1999. 48, 73
- [15] A. T. Campbell, S. B. Eisenman, N. D. Lane, E. Miluzzo, R. A. Peterson, H. Lu, X. Zheng, M. Musolesi, K. Fodor, and G. S. Ahn. The rise of people-centric sensing. *IEEE Internet Computing*, 12(4):12–21, July 2008. 14
- [16] Donald T Campbell and Julian C Stanley. Experimental and quasi-experimental designs for research. *Handbook of research on teaching (NL Gage, Ed.)*, pages 171–246, 1966. 145
- [17] Matt Carkci. Dataflow and reactive programming systems. *Create Space Independent Publishing Platform*, 2014. 112
- [18] John Carroll, Michael Horning, Blaine Hoffman, Craig Ganoë, Harold Robinson, and Mary Beth Rosson. Visions, participation and engagement in new community information infrastructures. *The Journal of Community Informatics*, 7(3), 2011. 31
- [19] Pew Research Center. Smartphone ownership and internet usage continues to climb in emerging economies, 2016. 12

- [20] Delphine Christin. Privacy in mobile participatory sensing: Current trends and future challenges. *Journal of Systems and Software*, 116:57 – 68, 2016. 16
- [21] Citclops. Citizens’ observatory for coast and ocean optical monitoring. <http://www.citclops.eu/>. Accessed: 2017-01-17. 28
- [22] CITI-SENSE. Development of sensor-based citizens’ observatories community for improving the quality of life in cities. <http://www.citi-sense.eu/>. Accessed: 2017-01-17. 28
- [23] COBWEB. Citizen observatory web. <https://cobwebproject.eu/>. Accessed: 2017-01-17. 28
- [24] European Commission. Green paper on citizen science for europe: Towards a society of empowered citizens and enhanced research, 2014. 14
- [25] Socket.IO Contributors. Socket.io. <https://socket.io/>. Accessed: 2017-10-13. 117
- [26] E. De Cristofaro and C. Soriente. Participatory privacy: Enabling privacy in participatory sensing. *IEEE Network*, 27(1):32–36, January 2013. 20
- [27] David M. Aanensen, Derek M. Huntley, Edward J. Feil, Fada’a al Own, and Brian G. Spratt. EpiCollect: Linking Smartphones to Web Applications for Epidemiology, Ecology and Community Data Collection. *PLoS ONE*, 4(9), 2009. 2, 18
- [28] David M. Aanensen, Derek M. Huntley, Mirko Menegazzo, Chris I. Powell, and Brian G. Spratt. Epicollect+: linking smartphones to web applications for complex data collection projects. *F1000Research*, 3:199, 2014. 18
- [29] AL Davis. Data driven nets—a class of maximally parallel, output-functional program schemata. *Burroughs IRC Report, San Diego*, 1974. 49
- [30] AL Davis and SA Lowder. A sample management application program in a graphical data driven programming language. *Digest of Papers Compton Spring*, 81:162–167, 1981. 49
- [31] Alan L Davis. The architecture and system method of ddm1: A recursively structured data driven machine. In *Proceedings of the 5th annual symposium on Computer architecture*, pages 210–215. ACM, 1978. 44

Bibliography

- [32] Delphine Christin, Andreas Reinhardt, Salil S. Kanhere, and Matthias Hollick. A survey on privacy in mobile participatory sensing applications. *Journal of Systems and Software*, 84(11):1928–1946, 2011. 14, 16, 17, 27
- [33] Tamara Denning, Adrienne Andrew, Rohit Chaudhri, Carl Hartung, Jonathan Lester, Gaetano Borriello, and Glen Duncan. Balance: towards a usable pervasive wellness application with accurate activity inference. In *Proceedings of the 10th workshop on Mobile Computing Systems and Applications*, page 5. ACM, 2009. 16
- [34] Jack B Dennis and David P Misunas. A preliminary architecture for a basic data-flow processor. In *ACM SIGARCH Computer Architecture News*, volume 3, pages 126–132. ACM, 1975. 44
- [35] Srinivas Devarakonda, Parveen Sevusu, Hongzhang Liu, Ruilin Liu, Liviu Iftode, and Badri Nath. Real-time air quality monitoring through mobile sensing in metropolitan areas. In *Proceedings of the 2nd ACM SIGKDD international workshop on urban computing*, page 15. ACM, 2013. 12
- [36] Ellie D’Hondt, Jesse Zaman, Eline Philips, Elisa Gonzalez Boix, and Wolfgang De Meuter. Orchestration support for participatory sensing campaigns. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp ’14, pages 727–738, New York, NY, USA, 2014. ACM. 3, 21, 23, 26, 30
- [37] Herbertt BM Diniz, Emanuel CGF Silva, and Kiev Santos da Gama. A reference architecture for a crowdsensing platform in smart cities. In *Proceedings of the annual conference on Brazilian Symposium on Information Systems: Information Systems: A Computer Socio-Technical Perspective*, 2015. 36
- [38] NP Edwards. The effect of certain modular design principles on testability. In *ACM SIGPLAN Notices*, volume 10, pages 401–410. ACM, 1975. 45
- [39] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G-S. Ahn, and A. T. Campbell. The bikenet mobile sensing system for cyclist experience mapping. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys ’07, pages 87–101, New York, NY, USA, 2007. ACM. 16
- [40] Shane B Eisenman, Nicholas D Lane, Emiliano Miluzzo, Ronald A Peterson, Gahng-Seop Ahn, and Andrew Campbell. Metrosense project: People-centric

- sensing at scale. In *Workshop on World-Sensor-Web (WSW 2006)*, Boulder. Citeseer, 2006. 16
- [41] Ellie D’Hondt, Matthias Stevens, and An Jacobs. Participatory noise mapping works! An evaluation of participatory sensing as an alternative to standard techniques for environmental monitoring. *Pervasive and Mobile Computing*, 9(5):681–694, 2013. 2, 20, 92, 137, 157
- [42] Eric Paulos. Citizen Science: Enabling Participatory Urbanism. In Marcus Foth, editor, *Handbook of Research on Urban Informatics: The Practice and Promise of the Real-Time City*, chapter 28, pages 414–436. Information Science Reference, IGI Global, 2009. 12, 13
- [43] Denzil Ferreira, Vassilis Kostakos, and Anind K Dey. Aware: mobile context instrumentation framework. *Frontiers in ICT*, 2:6, 2015. 19
- [44] Inc. FitnessKeeper. Runkeeper. <https://play.google.com/store/apps/details?id=com.fitnesskeeper.runkeeper.pro>. Accessed: 2016-09-29. 16
- [45] FixMyStreet. FixMyStreet Website. <https://www.fixmystreet.com/>. Accessed: 2016-09-30. 3, 17
- [46] Flowhub. Full-stack visual programming at your fingertips. <https://flowhub.io/>. Accessed: 2017-08-08. 51
- [47] Blender Foundation. Blender. <https://www.blender.org/>. Accessed: 2017-07-25. 53
- [48] Node.js Foundation. Node.js. <https://nodejs.org/en/>. Accessed: 2017-08-01. 123
- [49] Raghu K Ganti, Fan Ye, and Hui Lei. Mobile crowdsensing: current state and future challenges. *IEEE Communications Magazine*, 49(11):32–39, 2011. 14
- [50] Hui Gao, Chi Harold Liu, Wendong Wang, Jianxin Zhao, Zheng Song, Xin Su, Jon Crowcroft, and Kin K Leung. A survey of incentive mechanisms for participatory sensing. *IEEE Communications Surveys & Tutorials*, 17(2):918–943, 2015. 19
- [51] S. Gao, J. Ma, W. Shi, G. Zhan, and C. Sun. Trpf: A trajectory privacy-preserving framework for participatory sensing. *IEEE Transactions on Information Forensics and Security*, 8(6):874–887, June 2013. 20

Bibliography

- [52] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, February 1992. 45
- [53] Adele Goldberg, Margaret Burnett, and Ted Lewis. Visual object-oriented programming. chapter What is Visual Object-oriented Programming?, pages 3–20. Manning Publications Co., Greenwich, CT, USA, 1995. 73
- [54] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983. 72
- [55] David Hasenfratz, Olga Saukh, Silvan Sturzenegger, and Lothar Thiele. Participatory air pollution monitoring using smartphones. *Mobile Sensing*, pages 1–5, 2012. 17
- [56] George T. Heineman and William T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 46, 96, 99
- [57] John Hicks, Nithya Ramanathan, Donnie Kim, Mohamad Monibi, Joshua Selsky, Mark Hansen, and Deborah Estrin. Andwellness: an open mobile system for activity and experience sampling. In *Wireless Health 2010*, pages 34–43. ACM, 2010. 16
- [58] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Hum.-Comput. Interact.*, 1(4):311–338, December 1985. 73
- [59] Adobe Systems Inc. Phonegap. <https://phonegap.com/>. Accessed: 2018-02-20. 159
- [60] Facebook Inc. React - a javascript library for building user interfaces. <https://reactjs.org/>. Accessed: 2018-03-16. 47
- [61] Meteor Development Group Inc. Building apps with javascript | meteor. <https://www.meteor.com/>. Accessed: 2017-08-01. 47
- [62] National Instruments. Labview. <http://www.ni.com/en-us/shop/labview.html>. Accessed: 2017-07-25. 53
- [63] Native Instruments. Reaktor. <https://www.native-instruments.com/en/products/komplete/synths/reaktor-6/>. Accessed: 2017-07-25. 53

- [64] iSPEX. Measure aerosols with your smartphone, 2013. <http://ispex.nl/en>. 2, 20
- [65] Jeffrey A. Burke, Deborah Estrin, Mark Hansen, Andrew Parker, Nithya Ramanathan, Sasank Reddy, and Mani B. Srivastava. Participatory Sensing. In *WSW'06: Workshop on World-Sensor-Web, held at ACM SenSys '06*, 2006. 1, 13, 14, 20, 21
- [66] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004. xi, 44, 45, 49
- [67] jsPlumb Pty Ltd. Jsplumb. <https://jsplumbtoolkit.com/>. Accessed: 2017-10-13. 121
- [68] Salil S Kanhere. Participatory sensing: Crowdsourcing data from mobile smartphones in urban spaces. In *International Conference on Distributed Computing and Internet Technology*, pages 19–26. Springer, 2013. 21
- [69] E. Kanjo, J. Bacon, D. Roberts, and P. Landshoff. Mobsens: Making smart phones smarter. *IEEE Pervasive Computing*, 8(4):50–57, Oct 2009. 17
- [70] Eiman Kanjo. Noisespy: A real-time mobile phone platform for urban noise monitoring and mapping. *Mobile Networks and Applications*, 15(4):562–574, 2010. 17
- [71] Ryoma Kawajiri, Masamichi Shimosaka, and Hisashi Kahima. Steered crowdsensing: Incentive design towards quality-oriented place-centric crowdsensing. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '14, pages 691–701, New York, NY, USA, 2014. ACM. 19
- [72] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008. 51
- [73] A. Khan, S. K. A. Imon, and S. K. Das. Ensuring energy efficient coverage for participatory sensing in urban streets. In *2014 International Conference on Smart Computing*, pages 167–174, Nov 2014. 19
- [74] Sunyoung Kim, Jennifer Mankoff, and Eric Paulos. Sensr: Evaluating a flexible framework for authoring mobile data-collection tools for citizen science. In

Bibliography

- Proc. Computer Supported Cooperative Work*, pages 1453–1462. ACM, 2013. 2, 18
- [75] Paul R. Kosinski. A data flow language for operating systems programming. *SIGPLAN Not.*, 8(9):89–94, January 1973. 44
- [76] Andreas Krause, Eric Horvitz, Aman Kansal, and Feng Zhao. Toward community sensing. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks, IPSN '08*, pages 481–492, Washington, DC, USA, 2008. IEEE Computer Society. 14
- [77] LandSense. Landsense citizen observatory. <https://landsense.eu/>. Accessed: 2017-12-29. 28
- [78] Nicholas D. Lane, Shane B. Eisenman, Mirco Musolesi, Emiliano Miluzzo, and Andrew T. Campbell. Urban sensing systems: Opportunistic or participatory? In *Proceedings of the 9th Workshop on Mobile Computing Systems and Applications, HotMobile '08*, pages 11–16, New York, NY, USA, 2008. ACM. 14
- [79] Nicholas D Lane, Mu Lin, Mashfiqui Mohammad, Xiaochao Yang, Hong Lu, Giuseppe Cardone, Shahid Ali, Afsaneh Doryab, Ethan Berke, Andrew T Campbell, et al. Bewell: Sensing sleep, physical activities and social interactions to promote wellbeing. *Mobile Networks and Applications*, 19(3):345–359, 2014. 16
- [80] Nicholas D. Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T. Campbell. A survey of mobile phone sensing. *Comm. Mag.*, 48(9):140–150, 2010. 13
- [81] Vitaveska Lanfranchi, N Ireson, U When, SN Wrigley, and C Fabio. Citizens' observatories for situation awareness in flooding. In *Proceedings of the 11th International Conference on Information Systems for Crisis Response and Management (ISCRAM 2014): 18-21 May 2014*, pages 145–154, 2014. 3, 28
- [82] Juong-Sik Lee and Baik Hoh. Dynamic pricing incentive for participatory sensing. *Pervasive and Mobile Computing*, 6(6):693 – 708, 2010. Special Issue PerCom 2010. 19
- [83] Mattias Linnap and Andrew Rice. Managed participatory sensing with yousense. *Journal of Urban Technology*, 21(2):9–26, 2014. 19

- [84] Hai-Ying Liu, Mike Kobernus, David Broday, and Alena Bartonova. A conceptual approach to a citizens' observatory – supporting community-based environmental governance. *Environmental Health*, 13(1):107, 2014. 2, 26, 28, 29
- [85] Vinit Lohan. Component based effort estimation during software development: problematic view. *IJCSMS International Journal of Computer Science and Management Studies*, 11(03), 2011. 46
- [86] Hong Lu, Denise Frauendorfer, Mashfiqui Rabbi, Marianne Schmid Mast, Gokul T Chittaranjan, Andrew T Campbell, Daniel Gatica-Perez, and Tanzeem Choudhury. Stresssense: Detecting stress in unconstrained acoustic environments using smartphones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 351–360. ACM, 2012. 16
- [87] Hong Lu, Wei Pan, Nicholas D Lane, Tanzeem Choudhury, and Andrew T Campbell. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, pages 165–178. ACM, 2009. 17
- [88] Pattie Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987. 32
- [89] Nicolas Maisonneuve, Matthias Stevens, and Bartek Ochab. Participatory noise pollution monitoring using mobile phones. *Information Polity*, 15(1, 2):51–71, 2010. 12, 17, 22, 25
- [90] Ines Mergel. Distributed democracy: Seeclickfix.com for crowdsourced issue reporting. *Com for Crowdsourced Issue Reporting (January 27, 2012)*, 2012. 17
- [91] MicroFLo. Flow-based programming for microcontrollers (arduino++). <https://github.com/microflo/microflo>. Accessed: 2017-08-08. 47, 51
- [92] Prashanth Mohan, Venkata N. Padmanabhan, and Ramachandran Ramjee. Ner-cell: Rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, SenSys '08*, pages 323–336, New York, NY, USA, 2008. ACM. 17
- [93] J Paul Morrison. C# implementation of flow-based programming (fbp). <https://github.com/jpaulm/csharpfbp>. Accessed: 2017-08-08. 47

Bibliography

- [94] J Paul Morrison. Fbp vs. fbp-inspired systems. <http://www.jpaulmorrison.com/fbp/noflo.html>. Accessed: 2017-12-10. 43
- [95] J Paul Morrison. Java implementation of flow-based programming (fbp). <https://github.com/jpaulm/javafbp>. Accessed: 2017-08-08. 47
- [96] J Paul Morrison. Flow-based programming. In *Proc. 1st International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 25–29, 1994. 45, 46
- [97] J. Paul Morrison. Flow-based programming. *Journal of Application Developers' News*, 1, 2013. 43
- [98] MsgFlo. Flow-based programming with message queues. <https://github.com/msgflo/msgflo>. Accessed: 2017-08-08. 51, 58
- [99] Mohamed Musthag, Andrew Raij, Deepak Ganesan, Santosh Kumar, and Saul Shiffman. Exploring micro-incentive strategies for participant compensation in high-burden studies. In *Proceedings of the 13th International Conference on Ubiquitous Computing, UbiComp '11*, pages 435–444, New York, NY, USA, 2011. ACM. 19
- [100] D. Méndez, A. J. Pérez, M. A. Labrador, and Juan José Marrón. P-sense: A participatory sensing system for air pollution monitoring and control. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on*, pages 344–347, March 2011. 17
- [101] Inc. Nike. Nike+ run club. <https://play.google.com/store/apps/details?id=com.nike.plusgps>. Accessed: 2016-09-29. 16
- [102] Node-RED. Flow-based programming for the internet of things. <https://nodered.org/>. Accessed: 2017-08-08. 47, 51
- [103] GROW Observatory. Grow - build better soil through citizen science. <https://growobservatory.org/>. Accessed: 2017-12-29. 28
- [104] Omniscientis. Odour monitoring and information system based on citizen and technology innovative sensors. <http://www.omniscientis.eu/>. Accessed: 2017-01-17. 28
- [105] Eric Paulos, Richard J Honicky, and Elizabeth Goodman. Sensing atmosphere. *Human-Computer Interaction Institute*, page 203, 2007. 14

- [106] Bratislav Predic, Zhixian Yan, Julien Eberle, Dragan Stojanovic, and Karl Aberer. Exposuresense: Integrating daily activities with air quality using mobile participatory sensing. In *PerCom Workshops*, 2013. 16
- [107] Rajib Kumar Rana, Chun Tung Chou, Salil S Kanhere, Nirupama Bulusu, and Wen Hu. Ear-phone: an end-to-end participatory urban noise mapping system. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 105–116. ACM, 2010. 17
- [108] Sasank Reddy, Deborah Estrin, Mark H. Hansen, and Mani B. Srivastava. Examining micro-payments for participatory sensing data collections. In *UbiComp*, 2010. 19
- [109] Sasank Reddy, Andrew Parker, Josh Hyman, Jeff Burke, Deborah Estrin, and Mark Hansen. Image browsing, processing, and clustering for participatory sensing: lessons from a dietsense prototype. In *Proceedings of the 4th workshop on Embedded networked sensors*, pages 13–17. ACM, 2007. 16
- [110] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, November 2009. xi, 39, 48
- [111] John P. Rula and Fabián E. Bustamante. Crowd (soft) control: moving beyond the opportunistic. In Gaetano Borriello and Rajesh Krishna Balan, editors, *HotMobile*, page 3. ACM, 2012. 19
- [112] SCENT. Scent - smart toolbox for engaging citizens into a people-centric observation web. <https://scent-project.eu/>. Accessed: 2017-12-29. 28
- [113] Immanuel Schweizer, Roman Bärtl, Axel Schulz, Florian Probst, and Max Mühläuser. Noisemap-real-time participatory noise maps. In *Proc. 2nd Int'l Workshop on Sensing Applications on Mobile Phones (PhoneSense'11)*, pages 1–5, 2011. 12
- [114] SeeClickFix. SeeClickFix Website. <http://seeclickfix.com/>. Accessed: 2016-09-30. 17
- [115] Panos A. Ligomenides Shi-Kou Chang, Tadao Ichikawa, editor. *Visual Languages*. Plenum Press, New York, 1986. 48

Bibliography

- [116] B. Shneiderman. Direct manipulation: A step beyond programming languages. *Computer*, 16(8):57–69, Aug 1983. 74
- [117] Jonathan Silvertown. A new dawn for citizen science. *Trends in Ecology & Evolution*, 24(9):467 – 471, 2009. 14
- [118] Ingo Simonis and Rob Atkinson. Standardized Information Models to Optimize Exchange Reusability and Comparability of Citizen Science Data. Technical report, 2016. 64
- [119] Gurminder Singh and Mark H Chignell. Components of the visual computer: a review of relevant technologies. *The Visual Computer*, 9(3):115–142, 1992. xi, 48, 49
- [120] Matthias Stevens. *Community memories for sustainable societies: The case of environmental noise*. PhD thesis, Vrije Universiteit Brussel, 2012. 30
- [121] Erich P Stuntebeck, John S Davis II, Gregory D Abowd, and Marion Blount. Healthsense: classification of health-related sensor data through user-assisted machine learning. In *Proceedings of the 9th workshop on Mobile computing systems and applications*, pages 1–5. ACM, 2008. 16
- [122] Yu Sun, Zekai Demirezen, Marjan Mernik, Jeff Gray, and Barrett Bryant. Is my dsl a modeling or programming language? In *Domain-Specific Program Development*, page 4, 2008. 50
- [123] H. Tangmunarunkit, C. K. Hsieh, B. Longstaff, S. Nolen, J. Jenkins, C. Ketcham, J. Selsky, F. Alquaddoomi, D. George, J. Kang, Z. Khalapyan, J. Ooms, N. Ramanathan, and D. Estrin. Ohmage: A general and extensible end-to-end participatory sensing platform. *ACM Trans. Intell. Syst. Technol.*, 6(3):38:1–38:21, April 2015. 2, 18
- [124] Steven L. Tanimoto. Viva: A visual language for image processing. *J. Vis. Lang. Comput.*, 1(2):127–139, June 1990. 74
- [125] Microsoft Open Technologies and RxJS contributors. Reactive-extensions/rxjs: The reactive extensions for javascript. <https://github.com/Reactive-Extensions/RxJS/>. Accessed: 2017-09-20. 47, 106
- [126] Sameer Tilak. Real-world deployments of participatory sensing applications: Current trends and future directions. *ISRN Sensor Networks*, 2013, 2013. 2, 14, 15

- [127] Yoshitaka Ueyama, Morihiko Tamai, Yutaka Arakawa, and Keiichi Yasumoto. Gamification-based incentive mechanism for participatory sensing. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on*, pages 98–103. IEEE, 2014. 19
- [128] I. J. Vergara-Laurens, D. Mendez, and M. A. Labrador. Privacy, quality of information, and energy consumption in participatory sensing systems. In *Pervasive Computing and Communications (PerCom), 2014 IEEE International Conference on*, pages 199–207, March 2014. 19
- [129] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelman, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. 50
- [130] Michael von Kaenel, Philipp Sommer, and Roger Wattenhofer. Ikarus: Large-scale participatory sensing at high altitudes. In *2th Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011. 2, 20
- [131] Waylay. Waylay - where iot & apis, rules and integration meet. <https://www.waylay.io/>. Accessed: 2017-08-08. 54
- [132] Kung Song Weng. *Stream-oriented computation in recursive data flow schemas*. Massachusetts Institute of Technology, Project MAC, 1975. 44
- [133] WeSenseIt. The citizens’ observatory of water. <http://wesenseit.eu/>. Accessed: 2017-01-17. 28
- [134] Wikipedia. Component-based software engineering — wikipedia, the free encyclopedia, 2017. 46
- [135] XOD. A visual programming language for microcontrollers. <https://xod.io/>. Accessed: 2017-08-08. 54
- [136] J. Zaman, E. D’Hondt, E. G. Boix, E. Philips, K. Kambona, and W. De Meuter. Citizen-friendly participatory campaign support. In *2014 IEEE International Conference on Pervasive Computing and Communication Workshops (PERCOM WORKSHOPS)*, pages 232–235, March 2014. 23
- [137] Jesse Zaman. Orchestrating participatory sensing campaigns with workflows. Master’s thesis, Vrije Universiteit Brussel, June 2013. 21, 22, 23, 157

Bibliography

- [138] Willian Zamora, Carlos T Calafate, Juan-Carlos Cano, and Pietro Manzoni. A survey on smartphone-based crowdsensing solutions. *Mobile Information Systems*, 2016, 2016. 36
- [139] Daqing Zhang, Haoyi Xiong, Leye Wang, and Guanling Chen. Crowdrecruiter: Selecting participants for piggyback crowdsensing under probabilistic coverage constraint. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '14, pages 703–714, New York, NY, USA, 2014. ACM. 19