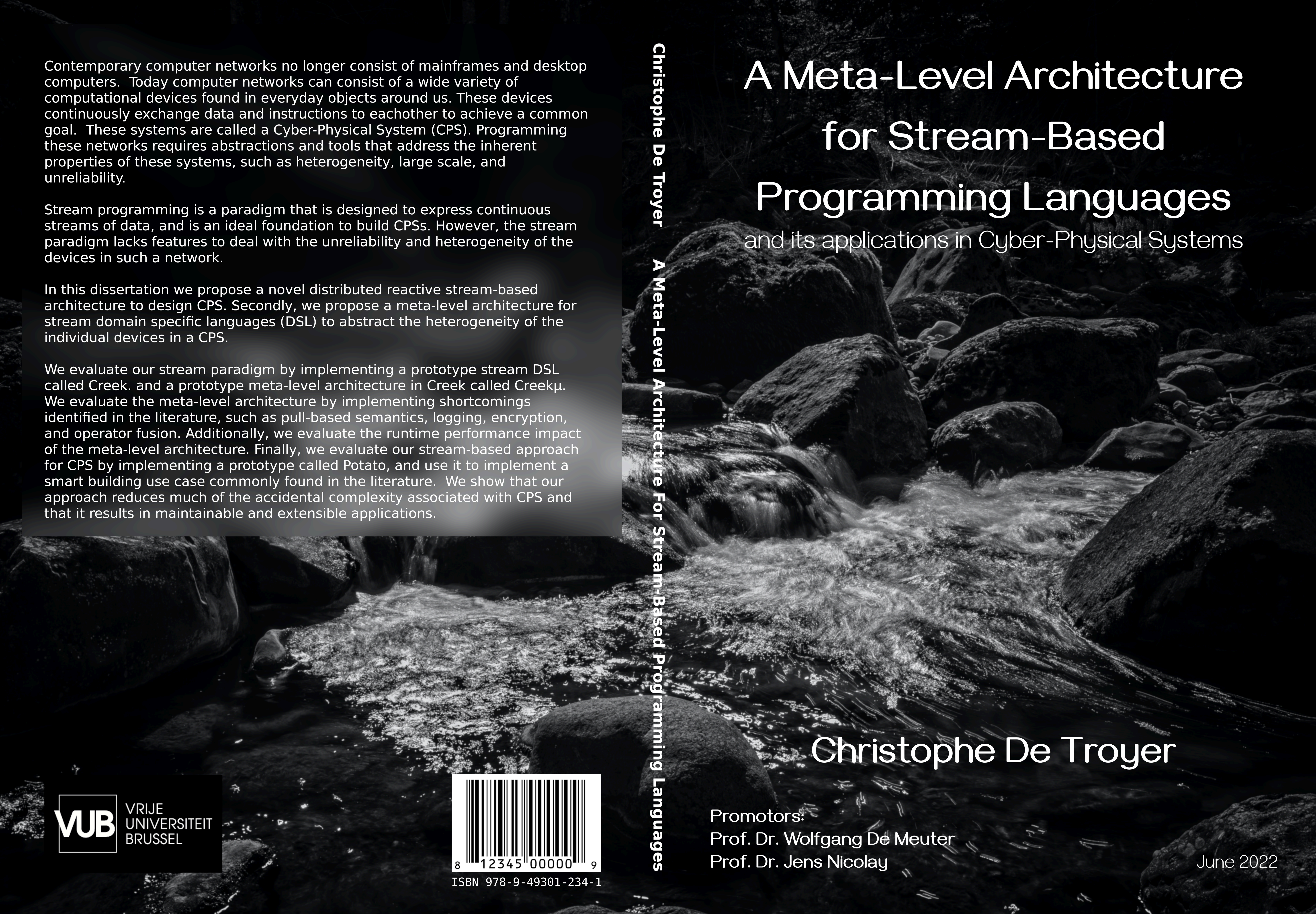


Contemporary computer networks no longer consist of mainframes and desktop computers. Today computer networks can consist of a wide variety of computational devices found in everyday objects around us. These devices continuously exchange data and instructions to eachother to achieve a common goal. These systems are called a Cyber-Physical System (CPS). Programming these networks requires abstractions and tools that address the inherent properties of these systems, such as heterogeneity, large scale, and unreliability.

Stream programming is a paradigm that is designed to express continuous streams of data, and is an ideal foundation to build CPSs. However, the stream paradigm lacks features to deal with the unreliability and heterogeneity of the devices in such a network.

In this dissertation we propose a novel distributed reactive stream-based architecture to design CPS. Secondly, we propose a meta-level architecture for stream domain specific languages (DSL) to abstract the heterogeneity of the individual devices in a CPS.

We evaluate our stream paradigm by implementing a prototype stream DSL called Creek. and a prototype meta-level architecture in Creek called Creekµ. We evaluate the meta-level architecture by implementing shortcomings identified in the literature, such as pull-based semantics, logging, encryption, and operator fusion. Additionally, we evaluate the runtime performance impact of the meta-level architecture. Finally, we evaluate our stream-based approach for CPS by implementing a prototype called Potato, and use it to implement a smart building use case commonly found in the literature. We show that our approach reduces much of the accidental complexity associated with CPS and that it results in maintainable and extensible applications.

VUB VRIJE UNIVERSITEIT BRUSSEL

8 12345 00000 9

ISBN 978-9-49301-234-1

Christophe De Troyer A Meta-Level Architecture For Stream-Based Programming Languages

A Meta-Level Architecture for Stream-Based Programming Languages

and its applications in Cyber-Physical Systems

Christophe De Troyer

Promotors:
Prof. Dr. Wolfgang De Meuter
Prof. Dr. Jens Nicolay

June 2022

A Meta-Level Architecture for Stream-Based Programming Languages and its Applications in Cyber-Physical Systems

Christophe De Troyer

Dissertation submitted in fulfillment of the
requirement for the degree of Doctor of Sciences

June, 2022

Promotors:

Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel
Prof. Dr. Jens Nicolay, Vrije Universiteit Brussel

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

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

ISBN 9789464443295
Nur Code 958
Thema UMA

If a man does not keep pace with his companions,
perhaps it is because he hears a different drummer.
Let him step to the music which he hears,
however measured or far away.

HENRY DAVID THOREAU, WALDEN

Abstract

Contemporary computer networks no longer consist of mainframes and desktop computers. Due to the miniaturization and mass production of computer chips, it has become feasible to embed computers into everyday objected, creating their "digital twin". The digital twin is causally connected to its physical counterpart; changes in the digital twin affect the physical device and vice versa. A network of digital twins is called a Cyber-Physical System (CPS). In a CPS, digital twins communicate to achieve a common goal and improve the user experience of the physical system. Examples of contemporary CPSs are smart factories, connected cars, oil pipeline monitoring, smart electricity grids, etc.

Contemporary CPSs consist of up to thousands of heterogeneous devices connected to the same network. Devices react to events and instructions emitted by other devices and emit new events and instructions in response, causing chain reactions throughout the system.

The different hardware and software traits of the devices affect the applications deployed on these networks. Some devices are battery-powered, making them unreliable in the long term. Some devices are connected using unreliable networks, making them unpredictable in the short term. Some devices are functionally equivalent (e.g., thermometers) but differ in their non-functional aspects (e.g., encodings or protocols), affecting how they integrate into the system. And finally, some devices are equipped with a computer that is not suited for complex calculations.

In this dissertation, we propose a two-pronged approach to design contemporary CPSs. First, we propose a novel distributed reactive stream-based architecture to design CPSs. Reactive stream applications are declarative transformations of possibly infinite streams of data. Additionally, distributed reactive streams allow multiple devices to work together on a single computation. The distributed stream-based paradigm maps well to the event-driven large-scale event-driven nature of CPSs. Secondly, we propose a meta-level architecture for stream domain specific languages (DSL) to express the aforementioned non-functional concerns of stream applications. We propose to use the meta-level architecture to express the non-functional concerns and heterogeneous software and hardware traits of the devices in a CPS.

We show that reactive streams is a valid paradigm to implement CPSs. The reactive, distributed nature of stream programming enables the design of scalable, distributed, resilient, event-driven systems. The paradigm nudges the programmer to design extensible applications that deal with open systems. The meta-level architecture of the stream language separates the non-functional and the logic necessary to integrate heterogeneous devices from the application

code.

We evaluate our stream paradigm by implementing a prototype stream Domain Specific Language (DSL) called Creek. We develop a prototype meta-level architecture in Creek called μCreek. We evaluate the meta-level architecture by implementing shortcomings identified in the literature, such as pull-based semantics, logging, encryption, and operator fusion. We evaluate the runtime performance impact of the meta-level architecture. We evaluate our stream-based approach for CPSs by implementing a prototype called Potato, and use it to implement a smart building use case commonly found in the literature. We show that our approach reduces much of the accidental complexity associated with CPSs and that it results in maintainable and extensible applications.

Abstract

Moderne computernetwerken bestaan niet langer enkel en alleen uit mainframes en computers. Dankzij de miniaturisatie en massaproductie van computerchips is het mogelijk geworden om kleine computerchips in dagdagelijkse objecten te stoppen, en zo een “digitale tweeling” te maken. Deze digitale tweeling is een virtuele causaal verbonden representatie van het object. Als er iets verandert in het object, verandert de digitale tweeling en omgekeerd. Een netwerk van digitale tweelingen noemen we een Cyber-Physical System (CPS). Apparaten in een CPS communiceren om een gezamenlijk doel te bereiken, en zo de gebruikerservaring van het gehele systeem te verbeteren. Moderne CPSen bestaan uit productiemachines, auto’s, oliepijpleidingen, robots, etc.

Moderne CPSen kunnen bestaan uit duizenden heterogene apparaten, verbonden op een gezamenlijk netwerk. Elk apparaat reageert op signalen en instructies van andere apparaten, en kan op zijn beurt nieuwe signalen en instructies op het netwerk sturen. Dit zorgt voor een kettingreactie doorheen het hele systeem.

De verschillende hard- en software eigenschappen van de heterogene apparaten in een CPS beïnvloeden op welke manier er software gemaakt wordt voor deze systemen. Sommige apparaten werken op een batterij, wat ze onbetrouwbaar maakt op de lange termijn. Sommige apparaten zijn verbonden met het netwerk door middel van onbetrouwbare verbindingen, wat ze onbetrouwbaar maakt op korte termijn. Sommige apparaten zijn functioneel equivalent (e.g., thermometers), maar verschillen in hun niet-functionele eigenschappen (e.g., data representatie of protocol). Deze verschillend zorgen ervoor dat apparaten niet uniform geïntegreerd kunnen worden in het netwerk. Ten laatste zijn er objecten wiens computer niet krachtig genoeg is om complexe berekeningen uit te voeren.

In deze verhandeling stellen we een tweeledige aanpak voor om moderne CPSen te ontwikkelen. Als eerste luik stellen we een nieuwe gedistribueerde event-driven architectuur voor op basis van reactieve stromen. Reactieve stroom applicaties zijn gedefinieerd als declaratieve transformaties van potentieel oneindige stromen van data. Gedistribueerde reactieve stromen maken het mogelijk voor meerdere apparaten om samen te werken aan een gezamenlijke berekening. Gedistribueerde stromen zijn uitermate geschikt om event-driven gedistribueerde systemen zoals CPSen uit te drukken. In het tweede luik van onze aanpak stellen we een meta-architectuur voor stromen voor om de non-functionele verschillen van apparaten te scheiden van de applicatie.

We tonen aan dat reactieve stromen een goed paradigma zijn om CPSen te implementeren. De reactieve en gedistribueerde eigenschappen van reactieve

stromen maken het mogelijk om schaalbare, gedistribueerde, veerkrachtige event-driven systemen te ontwerpen. Het paradigma begeleidt de ontwikkelaar om uitbreidbare en open systemen te ontwikkelen. De meta-level architectuur van de stromen scheidt de niet-functionele eigenschappen van de apparaten van de applicatie logica.

We evalueren ons stroom paradigma door een prototype stroom DSL te implementeren genaamd Creek. We ontwikkelen een prototype meta-level architectuur voor Creek, genaamd μCreek. We evalueren onze architectuur hiermee door tekortkomingen geïdentificeerd door de literatuur te implementeren zoals pull semantiek, logging, encryptie, en operator fusie. Verder evalueren we de performantie impact van onze meta-level architectuur. Tenslotte evalueren we onze CPS architectuur aan de hand van een prototype implementatie, genaamd Potato. We gebruiken Potato om een smart-building use-case uit de literatuur te implementeren. We tonen aan dat onze aanpak de accidentele complexiteit reduceert die gepaard gaat met CPSen, en resulteert in onderhoudsvriendelijke en uitbreidbare applicaties.

Acknowledgements

I would like to start these acknowledgments by thanking my promotors, Wolfgang De Meuter and Jens Nicolay. Jens, thank you for always listening to my ramblings and almost always doing an honest effort of trying to make sense of them. I hereby relieve you of your duty. Wolf, thank you for having insightful comments and ideas that helped shape the work presented in this dissertation, and most importantly, for allowing me to get a Ph.D. degree.

I would like to thank the members of my jury, in no particular order, Prof. Dr. Elisa Gonzalez Boix, Prof. Dr. Walter Cazolla, Prof. Dr. Hidehiko Masuhara, Prof. Dr. Ann Nowé, and Prof. Dr. Abdellah Touhafi. Thank you for taking the time to read my dissertation and the insightful questions.

More than anyone, I want to thank my wife, Kimber, for always believing in me and pushing me to do my best and persevere when the going got tough. Apparently, I wasn't always the most enjoyable person to be around in the past years, but that might be fake news, I'm adorable. Without your support and encouragement, I would certainly not be where I am today, and for that, I am forever grateful. One day you will get that statue in the town square of Aalst – the capital of the observable universe – which I think you truly deserve. I would also like to thank myself for finishing one of the hardest things I ever had to do.

During my career at the VUB, I have met many interesting people that I would like to express gratitude to. Maarten, thanks for being your weird self, and especially thanks for letting me rant every now and then. Florian "Smalle" Myter, thanks for always making me look swole when I stand next to you, and thanks for the many politically correct laughs we shared in the Church of Iron. Arnold is Numero Uno. Tall Maarten, even though we only worked together briefly, and you are a quitter, I'm glad I can count a true altruist amongst my friends. Quentin, thanks for going on smoke breaks with me, and for the many conversations we had about plants, economics, computer science, and countless other topics.

A word of gratitude is also necessary for all the people in the IRC channel `#infogroep`. The channel is filled with some of the smartest people I know, and all of them have always been helpful in one way or another, and have always been kind to me. A big thanks to, sorted in lexicographic order, `Botje`, `ErgRbt`, `Ian`, `Ichabod`, `inspiran`, `Numlock`, `omega`, `robbe`, `rubdos`, and `ward`.

Finally, I want to thank my family. Ma, pa, without your support I would not have had the chance to be where I am today. Dirk, Giselle, thanks for cheering me on all these years. I promise I'm getting a job now, and yes, yes, it won't be as fun as doing a Ph.D.

- Christophe

Dankwoerd

'K zo geeren men begeloiders bedanken, de Wolf en de Jens. Zemmen eir best gedoon ver te loisteren en mé te begroipen azzek gediereg ont afloeipen was oever 't iejn of 't ander. Ik peis da ze dikkes gepeist emmen in eir oigen: "wa nen destereer." Mor allei kom, weir zen der gerokt zonder al teveel kweddelen. Zonder eir zollek et gralek vermoeist emmen, echt woor, dikke merci.

Ik moeng die mensjen van menne jury oeik noig bedanken, want 't zen dedie die 't oon eere resaur oin ver giejl den bataklang te moeten leizen. Ni just teis ier, wa da ge goi na ont leizen zet. Prof. Dr. Elisa Gonzalez Boix, Prof. Dr. Walter Cazolla, Prof. Dr. Hidehiko Masuhara, Prof. Dr. Ann Nowé, en Prof. Dr. Abdellah Touhafi, dikke merci!

Kzal men vra, Kimber, oeik mor bedanken, of 'k gon paretten emmen tois. Binsjt dak men pansj zitten afdroin ém oon da verrozjeken ier, ejje goi ('t schantj toch, allei) me nen onverdraugelekken op a dek gezeiten. Een ander zol et ni gedoon emmen, mor goi wel. Zonder a mochten ze moi e zottekliejd oondoeng en binnesteiken, en oik zeiker giejnen doktoer geworren. Vazeleeven goje e stambeltj kroigen op de groeite mert, want achter dees histoere ejjet miejr as verdindj.

On aal de mensjen van ier in Brissel, merci vér alles. Kem dikkes gelachen, veel geliejrd, en veel kamerooten gemokt. 't Es zonne da da leste betjen cultuur na vertrekt oit Brissel, mor 't hangt ier men voetn oit en 'k ben den of af. Merci allemool, noste kiejr est bé ons.

As leste moenek mé ma en mé pa toch oeik bedanken. Azzek jonk was emmek er dikkes men voetn oon geveegd en tei vriejt lank gedierd, mor 'k ben der toch gerokt. Merci vér de possensje. Dirk en Giselle, 'k wil eir toch oeik bedanken ver azoei te supporteren. Da zejje oeik wel moei zen na. Merci ein.

Allei, kert en goed, 't es enjelek gedoon me schoefelen op eir kosten, plasjt isj in eir annen. Na kennen ze met'nienen iet nuttig doeng me eer belastingen. 't Was petank giejsteg vér den toid dat gedierd eit.

- Christophe

Contents

Acronyms

CPS	Cyber-Physical System.
CSP	Canonical Stream Protocol.
DAG	Directed Acyclic Graph.
DSL	Domain Specific Language.
IoT	Internet of Things.
WSAN	Wireless Sensor and Actuator Network.
WSN	Wireless Sensor Network.

List of Listings

Chapter 1

Introduction

Cyber Physical Systems (CPSs) are the integration of physical systems and computational processes. Physical devices are augmented with embedded computers that represent the state of the physical device and allow remote control of the physical device by means of software. The embedded computers are networked, making it possible to exchange data and commands. The software that runs on top of these networks is built to increase the efficacy of the physical systems. CPSs have been applied to increase response time in oil pipeline leaks [13], improve traffic flow at intersections [53], improve the sustainability and energy efficiency of large office spaces [50], and improve the energy distribution between households [5].

Designing CPS applications that run on top of these networks poses a unique set of software design challenges that have to be addressed by these application. CPSs are *large-scale* systems that can consist of hundreds of *heterogeneous devices*. Integrating many heterogeneous devices forces the application logic to cater to their different traits, polluting the application logic with *non-functional concerns* and hindering extensibility. Some devices participating in a CPS can be *limited in their computational power*, impacting the types of services they can provide to the application. The only way for devices with limited resources to provide complex services is to involve other devices in their computations. Finally, devices in a CPS may join or leave the network at undefined times, which forces the application to either use equivalent devices, or wait for the network to be restored to continue operation.

Developers need the appropriate tools to express such applications. The individual challenges just outlined have been thoroughly researched, but our literature study (see section 4.6) leads us to conclude that there is no approach

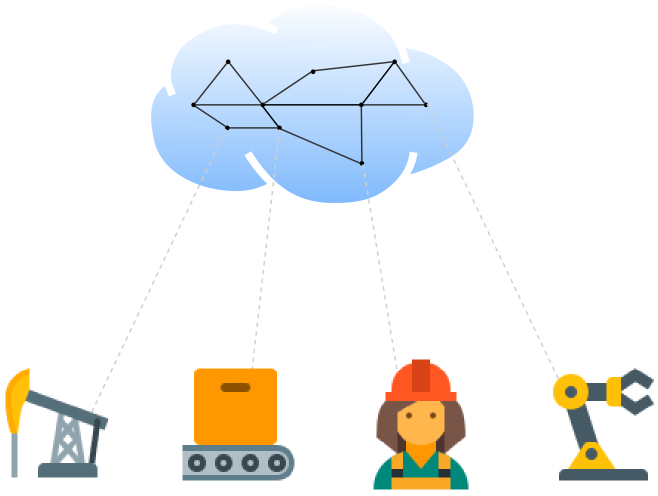


Figure 1.1: *Graphical Depiction of a CPS. All types of devices are connected to a network to exchange information and instructions.*

that addresses all the challenges in a unified paradigm, and no approach exists that uses the stream paradigm. *In brief, contemporary approaches do not address all these challenges, requiring the application developer to address them in an ad-hoc way, as part of the application code.*

At the heart of a CPS is a perpetual feedback loop, because it is an event-driven system connected to the physical world at its very core. CPS applications define data streams between devices, and the instructions that are sent to devices. *Reactive stream programming* is designed to express event-driven systems [54, 16] such as GUI programming [69], robotics [61], and data processing [96]. *In this dissertation we propose reactive streams as a foundation for CPS applications.*

The data exchange between devices can be trivially expressed as a reactive stream, however, the dynamic network evolution, network failures, and heterogeneous devices and their range of non-functional concerns are not directly translated to a stream-based approach. **In this work, we explore how we can incorporate these challenges in the stream paradigm.**

1.1 Towards Stream-Based CPSs

The reactive stream paradigm describes *what* happens to a stream of data, rather than *how*. Consider the example below of a stream of Kelvin temperature measurements. Two declarative transformations filter out all the measurements that are below 35 degrees Celcius.

```
thermometer
~> map(fn k -> k - 273.15 end)  # Kelvin to Celsius
~> filter(fn c -> c > 35.0 end) # Only values above 35.0 Celsius
```

The above stream definition describes the transformations that should happen (i.e., *what*) when the `thermometer` stream emits a new measurement. The above application is event-driven because the `thermometer` is an external process that triggers the computation. This *inversion of control* is inherent to event-driven systems, and is characteristic of CPSs as well. Traditionally, event-driven systems are programmed using *callbacks*. Callbacks contain a part of the application logic that is executed in reaction to an external event. The downside of callbacks is that application logic is spread over multiple callbacks, which reduces the program structure. Stream-based programming is an alternative to programming with callbacks. A stream declaratively specifies what happens in reaction to an event, while preserving the application logic structure.

We envision a programming framework where CPSs are **defined entirely as stream-based applications**. However, in the state of the art, not every challenge can be trivially addressed from within the stream paradigm. In what follows, we explain the design challenges of CPSs, and how we propose to tackle them in the stream paradigm. The result will be a framework where all concerns related to a CPS will be expressed in the stream paradigm.

1.1.1 Challenge 1: Hetereogeneous Devices

The devices in a CPS may represent a wide array of physical devices, resulting in a *network of heterogeneous devices*. The differences between these devices hinder integration into the application because the application has to cater to each device its specific properties. Additionally, the data streams between devices can have different *non-functional concerns*, such as security, performance, and quality of service. In software engineering, these non-functional concerns and differences between devices should be addressed separately from the application logic. Consider the case of two thermometers. Both thermometers measure their environmental temperature, but non-functionally, they can differ in the encoding of the values (e.g., XML vs. JSON), or the scale they use (e.g., Kelvin vs. Celcius).

Two popular approaches for dealing with non-functional concerns in applications are aspect-oriented programming [59, 60] and meta-level programming [20, 15, 42]. Research into meta-level programming has been done for object-oriented programming [15, 20], logic programming [19], and procedural languages [89]. To the best of our knowledge, no meta-programming approach exists for stream-based languages.

Conclusion The heterogeneity in a CPS requires application developers to adapt the application logic to address non-functional differences between devices. *Meta-level programming is a well-established approach to solve this problem, but no approaches exist for stream-based languages.*

1.1.2 Challenge 2: Large Scale

In practice, CPSs can contain hundreds of devices that each provide a service to the network, resulting in a complex web of communication channels between these devices and continuous streams of data. Manually defining pairwise communication in an application is not scalable; the approach becomes inconvenient beyond a few dozen devices. There is a need for abstractions that help to define a large number of communication channels without burdening the programmer to define them separately.

Streams excel at defining data transformations between devices, but offer little help in configuring transformations at a larger scale.

Conclusion The large-scale of CPSs burdens the programmer with setting up data dependencies between individual devices. Current stream-based frameworks can express many-to-many communication, but the streams between the devices need to be configured individually. *We propose scalable stream-based abstractions to designate devices and configure data streams between them.*

1.1.3 Challenge 3: Unreliable Systems

Some devices in a CPS are connected over an unreliable connection or have a limited power source, resulting in failures in parts of the application. As a result, the data streams defined between devices can be interrupted at undefined times. As a result, devices cannot rely on the availability of the services they consume, and interruptions must be handled by the application accordingly.

Contemporary stream libraries such as Akka Streams and RxJava feature a form of asynchronous failure handling to handle fatal application errors. However, we argue that a network failure in a CPS is not a fatal error, but a normal part of the application. This too, should be expressed in the stream paradigm.

Conclusion The unpredictable nature of CPS networks requires the application to deal with scenarios where a data stream is interrupted abruptly. We propose *network interruptions and device failures to be exposed at the application* *level as a stream of regular network events*, rather than failures, so the network failures can be addressed as part of the stream-based application logic.

1.1.4 Challenge 4: Open Systems

The network of a CPS is not static, because devices can join and leave the network at undefined times during the entire lifespan of the application. For example, mobile devices, broken devices, and replacements for broken devices, can cause changes in the topology of the network. When devices are part of a distributed service, the service breaks down when a device disconnects. The application has to define its data streams between devices in such a way that they deal with a changing network topology.

The stream paradigm in its original form does not provide any mechanisms to handle changes in the network topology. When part of a stream is unavailable, a program fault occurs. Services and their data streams should be defined such that a change in the network topology does not cause a program fault.

Conclusion The open nature of CPSs requires the application to define data streams between devices in terms of a dynamic network topology. *We propose a stream of network events that can be used to express applications that adapt to changes in the network.*

1.1.5 Challenge 5: Limited Computational Power

A CPS may contain devices that have limited computational power that cannot provide complex services on their own, but serve as a source of data in the system. Examples of such devices are thermometers, displays, and speakers. Creating services on these devices requires other devices to execute the complex computations.

Distributed stream programming can solve the challenge of involving other devices in a computation, but does not directly address the distribution of the computation on the network. It is unfeasible to provide all the necessary logic for a plethora of services when deploying a CPS application, especially in an open network. A dynamic approach is required that allows application logic to be distributed at runtime.

Conclusion Devices with limited resources in a CPS can provide services to the system if they involve other devices through distributed computations. Streaming data between devices on a network is not problematic in itself, however, deploying the necessary logic on the devices beforehand is required. *We* *propose to stream application logic (i.e., mobile descriptions of streams) to remote devices at runtime.*

1.2 Research Context

Our research is situated at the intersection of:

1. **Cyber-Physical Systems** Our research is centered around CPSs. As we have explained above, *we explore the applicability of the stream paradigm as a viable approach to express cyber-physical system applications.*

2. **Software Architecture** In this dissertation, we propose a stream-based software architecture for CPSs to tackle the challenges currently present in the design of CPSs. Devices and their data exchange can be naturally expressed as a data stream. We argue that network events and code deployment can be expressed in the stream paradigm, leading to *a unified programming paradigm to tackle all the challenges listed in section 1.1.*

3. **Meta Programming** The heterogeneous nature of the devices in a CPS creates variations in their non-functional concerns, which hinders integration. In this dissertation, *we define and implement a meta-level architecture for stream-based programming* and use it to address those variations.

1.3 Research Goal

This dissertation investigates how a stream-based approach can be used all the way down (i.e., application logic and non-functional concerns) to design CPSs. Streams are a promising paradigm to express CPSs, but additional research is necessary on how to address the challenges of CPSs. In particular, **we wish to represent all concepts of a CPS as a stream, i.e., the network, the data exchange between devices, and the non-functional concerns.** We impose the following requirements to our approach.

1. Data exchange between devices must be represented as a stream.

2. The open and volatile network must be represented as a stream.

3. Devices must be able to set up distributed streams at runtime.

4. The non-functional concerns of the devices must be expressed as a stream, separated from the application logic.

1.4 Approach

This dissertation proposes a novel stream-based approach to develop CPSs. We tackle this using two approaches.

1. *We distill the commonalities of contemporary stream Domain Specific Languages (DSLs)* based on a state of the art survey (section 2.1). *We implement a DSL prototype in a mainstream general-purpose programming language, called Creek, based on the survey (section 2.2.1).*

2. We *investigate how non-functional concerns are addressed by contemporary stream DSLs (section 3.2).*

3. *We present the design of a meta-level architecture for stream DSLs,* and build a prototype on top of our stream DSL called μCreek (section 3.4).

4. We *evaluate our meta-level architecture by implementing four non-functional concerns found in the literature (section 3.4.1).* Additionally, we implement a prototype debugger on top of the meta-level architecture to show its expressiveness (section 3.6).

5. *We propose a framework to express CPSs* in the stream paradigm, called Potato (section 4.3).

6. *We evaluate our framework by implementing a smart-building use case from literature.* We qualitatively compare our approach to related work (section 4.7).

1.5 Contributions

This dissertation makes the following contributions.

A stream-based framework for CPSs. We propose a novel stream-based framework called Potato to design CPS applications that represents all concepts from a CPS, such as the network, the data, and the devices as streams. Potato represents all concerns in a CPS to the programmer as a stream, allowing applications to deal with concerns such as scalability, service discovery, scalability and code offloading from within the stream paradigm.

A meta-level architecture for stream DSLs. A proposal for a meta-level architecture design for stream DSLs, called μCreek. The architecture is designed according to the design guidelines set forth by related work in meta-level design. μCreek makes it possible to address the heterogeneity of devices in a CPS. All the logic necessary to address the differences between devices is expressed as a meta-level application, rather than polluting the base level code.

Prototypical Implementations. We provide a prototypical implementation of a stream DSL, a meta-level architecture on top of this stream DSL, and a framework for CPSs based on these prototypes. All the artifacts are available online[1].

1.6 Supporting Publications

The work presented in this dissertation is published at peer-reviewed venues. We list and summarize them below.

- **Building Internet of Things (IoT) Systems using Distributed First-Class Reactive Programming** *Christophe De Troyer, Jens Nicolay, Wolfgang De Meuter* IEEE International Conference on Cloud Computing Technology and Science, 2018.

  This paper discusses the approach of the Potato middleware in the context of IoT systems. It introduces the stream-based paradigm around which Potato revolves, and shows it is well-suited to design large-scale distributed event-based systems.

- **The Art Of The Meta Stream Protocol** *Christophe De Troyer, Jens Nicolay, Wolfgang De Meuter* Art Science and Engineering of Programming, 3(3):5, 2021.

  This paper introduces meta-level facilities into the stream paradigm used in our other work [94]. The meta-level approach allows the programmer to define different semantics for specific streams in a single code base. We argue that a single application does not always have the intended semantics and that the meta-level is the best place to modify the semantics.

Additionally, two workshop papers were published that mark the beginning of our work presented in this dissertation [93, 92].

- **First-class reactive programs for CPS** *Christophe De Troyer, Jens Nicolay, Wolfgang De Meuter* Proceedings of the 4th ACM SIGPLAN Internal Workshop on Reactive and Event-Based Languages and Systems, 2017.

  This paper introduces the concept of stream-based programming for CPSs, and built on the hypothesis that CPSs are inherently event-driven systems that can be expressed as streams of data and software.

- **Abstractions for Distributed Event-Driven Applications**

[1] `http://github.com/softwarelanguageslab/creek`

Christophe De Troyer, Jens Nicolay, Wolfgang De Meuter, and Christophe Scholliers 2018 Companion to the first International Conference on the Art, Science and Engineering of Programming, 2017.

This paper proposes to equip every device in a CPS with a minimal virtual machine to execute application code that is deployed at runtime. The hypothesis being that moving code towards data improves resilience, and reduces network traffic.

Finally, this dissertation has lead to *five successfully defended master theses*.

1.7 Outline of the Dissertation

The dissertation is structured in the following chapters.

- **Chapter 2: Creek: A Prototypical Stream DSL**

  This chapter introduces our new stream language, called Creek. We use Creek as a technological vehicle to define what stream languages are, and as a yardstick to taxonomize related work.

- **Chapter 3: μCreek: A Meta-Programming Approach for Stream Languages**

  This chapter discusses our reflective architecture for stream-based languages, called μCreek. It introduces compile-time reflection, run-time reflection, and a performance evaluation of our reflective architecture compared to the language without reflective features. μCreek allows us to define the differences between devices in a CPS separately from the base-level application. This results in an application where non-functional and functional concerns are cleanly separated.

- **Chapter 4: Potato: A Streaming Platform for CPS**

  In this chapter, we discuss the design and features of Potato in detail. The chapter covers the reactive streams language and its core concepts, the representation of the network and entities as reactive streams, and first-class reactive programs.

- **Chapter 5: Implementation**

  In this chapter, we present a broad overview of our prototype implementation, Potato. We discuss the design and architecture of Potato, why we chose Elixir as a foundation for our prototype, and which of its features have benefited us. We explain the architecture of Potato, the mechanisms behind the representation of the network and devices, and the evaluation of the first-class reactive programs. We conclude with an overview of the meta-level architecture of the stream language.

- **Chapter 6: Conclusion**

  We conclude our dissertation by summarizing the contributions and the limitations of our work and discussing avenues for further research or improvement in the state of the art of CPS and reflective architectures for stream languages.

Chapter 2

Creek: A Prototypical Stream DSL

A Cyber-Physical System (CPS) is a system that consists of physical devices and their digital counterpart. The digital devices continuously generate events coming from their environment and react to events coming from other devices. The devices work together to create a user experience that is more useful than the sum of its parts. CPSs are *data-driven* systems, thus stream programming is well-suited to express them without the drawbacks associated with other event-driven paradigms.

Chapter 1 introduced software design challenges in the context of CPSs, and section 1.4 proposed stream programming as a suitable paradigm to tackle these challenges. Most contemporary general-purpose programming languages have a library or Domain Specific Language (DSL) that adds stream programming abstractions. The abstractions are highly similar across languages in terms of syntax but have – sometimes subtle – differences in their semantics. We taxonomize the state of the art, and based on that, we build a prototype stream DSL, called Creek. We use it as a basis to formulate a stream-based meta-level architecture in the next chapter (chapter 3). Finally, chapter 4 introduces Potato, a stream-based framework using Creek to program CPSs.

2.1 Stream DSL Properties

Many stream Domain Specific Languages (DSLs) have their own specific terminology for similar concepts, making it harder to compare features across DSLs. In this section, we use the terminology from Akka Streams and use it to identify equivalent concepts in other DSLs. We compare these concepts across the state of the art in stream DSLs. We selected DSLs for reactive streams in the most popular general purpose languages. However, all implementations of Reactive Extensions (i.e., RxJava, RxJS, RxRust, RxPython, ...) are modeled after the prototypical implementation, called RxJava, so we consider them the same and omit them. Akka Streams is the only DSL that is *two-phase*, where the definition of a stream and its execution (i.e., *deployment*) are distinct steps in the application. PLINQ and Java Streams are two DSLs that are part of their host language, C# and Java respectively. We included both in our comparison because PLINQ supports concurrency out of the box, while Java Streams does not. Finally, Flow, a DSL part of the Elixir language, is included because it is, to the best of our knowledge, the only *push-based* stream DSL. The summary of our literature study is shown in table 2.1.

Name	Execution	Phases	Semantics	Backpressure
RxJava	Concurrent	Single	Pull	Yes
Akka Streams	Concurrent	Double	Pull	Yes
Java Streams	Sequential*	Single	Pull	-
PLINQ	Concurrent*	Single	Pull	-
Flow	Concurrent	Single	Pull	Yes
Streamz	Sequential*	Single	Pull	-
Creek	Concurrent	Double	Push	No

Legend *–: Not Applicable, *: With option to modify.*

Table 2.1: *List of state of the art in streaming DSLs.*

Common Concepts At a high level, all stream DSLs revolve around four concepts. A stream is composed out of individual *operators*, atomic transformations that propagate through the stream at runtime. The operators that inject data into a stream are called *sources*, and the operators that consume data from a stream are called *sinks*. Operators are connected by their in- and outputs, forming a *DAG*. A Directed Acyclic Graph (DAG) is a blueprint for a *stream*. When a DAG is *deployed* data starts propagating through it. A deployed DAG is called a *stream*.

Single and Double Phase DSLs Exploring the state of the art in stream DSLs we find two main categories of DSLs: *single phase* and *double phase* DSLs. In the former, a stream is defined as an expression that is immediately evaluated. In double phase DSLs the definition of the DAG and its deployment are separate steps. In other words, a stream definition is a *first-class value* in double phase languages, typically called a *DAG*. DAGs can be assigned to variables, returned from, and passed to functions.

Double phase DSLs have two advantages over single phase DSLs. First of all, in double phase DSLs, streams can be composed out of separately defined DAGs. Single phase DSLs require the stream to be defined in a single expression. This limitation prohibits single phase DSLs to create partial streams, and hinders modularity.

The second advantage of double phase DSLs is that the stream, i.e., application logic, can be defined separately from its data sources and sinks. In a single phase DSL the data sources and sinks are part of the stream definition.

A double phase language makes it easier to write reusable and composable stream definitions. In the state of the art we surveyed, Akka Streams and Creek are both double phase DSLs, while RxJava, Java Streams, Flow, LINQ, and Streamz are single-phase DSLs.

```scala
object Factorial extends App {
  implicit val system: ActorSystem = ActorSystem("Factorial")
  val source = Source(1 to 5)
  val sink = Sink.head[Int]
  val factorial = Flow[Int].reduce((acc, e) => acc * e)
  val stream: Future[Int] = source.via(factorial).runWith(sink)
  val res: Int = Await.result(stream, Duration(5, TimeUnit.SECONDS))
  print(res)
}
```

Listing 1: *Composing source, sink, and transformations in Akka Stream.*

```java
public class Factorial {
    public static void main(String[] args) {
        Flowable<Integer> source = Flowable.range(1, 5);
        Flowable<Integer> operation = source.scan((acc, e) -> acc * e);
        Disposable result = operation
                            .takeLast(1)
                            .subscribe(System.out::println);
    }
}
```

Listing 2: *RxJava requires source and operations to be defined in a single expression.*

Consider the code listings in listings 1 and 2. The listings show how to calculate the factorial of 5 in RxJava (listing 2) and Akka Stream (listing 1). RxJava is a single phase DSL, and Akka Stream is a double phase DSL. This becomes apparent when defining the source, sink, and transformations of the stream.

In listing 1, the source (line 3), sink (line 4), and transformation (line 5) are defined separately and require no reference to each other. Composition is only required before the stream is *deployed* (i.e., executed). This is possible because the DAGs `source`, `sink`, and `factorial`, are first-class values. In listing 2, the source (line 3) is the only operator that can be defined separately from the stream. The transformations (line 4) and the sink (line 5) have to be defined in terms of their predecessors in the stream. Additionally, the sink definition (lines 5–7) doubles as the deployment step.

Execution A stream is – by construction – a pipeline with distinct *stages* or *operators*. Each operator is an atomic functional unit that applies a transformation to each datum propagating through the stream. Each operator takes as input the result of one or more preceding operators (*upstream operators*). Once an operator has processed a datum, it *emits* the resulting datum to its proceeding operators (*downstream operators*). Due to the atomic nature of operators, the operators can execute in *pipeline parallelism* [75]. Pipeline parallelism allows every operator to process a datum concurrently with the other operators. Looking at the state of the art, we see that some DSLs (e.g., Java Streams, RxJava, and PLINQ) execute streams sequentially by default but offer the programmer a way to introduce parallelism by explicitly annotating parts of the stream. Listing 3 shows an example of an RxJava stream where the `map` operator is parallelized. The highlighted lines indicate changes in the stream definition to enable parallelism. The parallelization is obtained by manually placing different parts of a stream onto separate (virtual) processes. Akka Streams, Flow, and Creek, execute streams using pipeline parallelism by default.

```
source
.subscribeOn(Schedulers.computation())
.flatMap(val ->
    just(val)
    .subscribeOn(Schedulers.computation())
    .map(i -> longLastingComputation(i))
    .subscribeOn(Schedulers.single()))
.subscribe();
```

Listing 3: *Parallelizing a stream in RxJava.*

Propagation Semantics If one operator of a stream is executing in a separate process, concurrency exists between operators. For example, a stream that depends on an third-party stream of data (e.g., a Twitter stream) is concurrent with its source. Concurrency can cause *rate mismatch* between a pair of operators. When the producer of a stream produces data at a faster rate than a consumer (e.g., a database), some operators in the stream will become *congested*. A congested operator cannot process data as fast as the upstream is providing it. The data generated by the producer must be buffered somewhere until the consumer can process it. An unresolved rate mismatch eventually leads to data loss. A solution to rate mismatch is changing the way data propagates through a stream, called the *propagation semantics*. The propagation semantics of a stream is the protocol operators use to communicate to propagate data through the stream. The simplest form of propagation semantics is called *push-based semantics*, where each operator receives data and emits it as fast as it can process them. It requires no additional communication between operators besides the data itself, but is susceptible to rate mismatch.

Figure 2.1 graphically depicts the push-based semantics of a stream between three operators. Operator A emits its values to B, which processes the values and propagates them to C. In this diagram, C is the slowest operator in the link, creating a rate mismatch between C and its upstream. If this is not resolved, messages emitted by B will be lost.

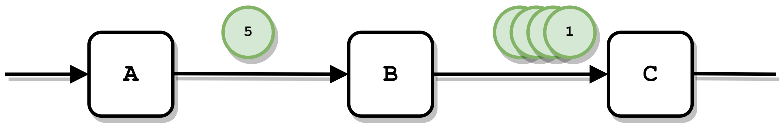


Figure 2.1: *Diagram depicting the communication for push-based semantics.*

The most common semantics in the state of the art is *backpressure propagation*, or a variation of backpressure propagation. Figure 2.2 graphically depicts a diagram depicting the simplest form of backpressure. When an operator notices, e.g., by looking at its message queue, that it can no longer process enough values to keep up with the upstream, it will signal its upstream to back off. When the operator has processed its queue, it can signal the upstream to send more data. Most stream DSLs (i.e., RxJava and Akka Streams) implement a variation of backpressure. In the state of the art, backpressure is typically only present in DSLs that execute streams concurrently. In a sequential stream DSL, only one stage is computing at any given time, ruling out rate mismatch entirely. PLINQ, Streamz, and Java Streams are examples where backpressure is absent. Akka Streams and Flow offer some form of backpressure because they have parallel execution.

Figure 2.2 graphically depicts backpressure between three operators. C is the slowest operator in the stream, and values are building up in its message queue. When the queue reaches some predefined threshold, the operator sends an internal message to its upstream operators to stop producing data ❶. B, all other operators propagate the request upstream ❷ until it reaches a source, which stops producing data.

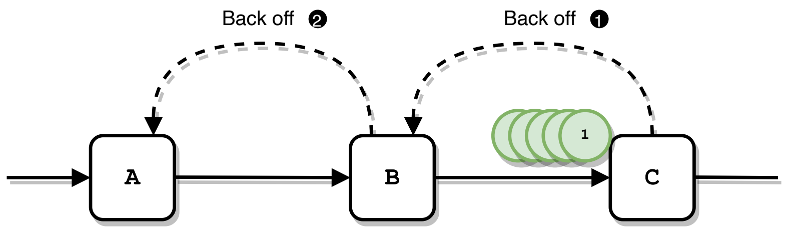


Figure 2.2: *Diagram of backpressure in a stream.*

2.2 Creek: A Novel Stream Programming DSL

Creek is a prototype implementation of a generic stream Domain Specific Language (DSL) that will serve as an experimentation vehicle for our approach in chapter 3. In this section we show a example application application, introduce Creek and its syntax, define the components of a Creek program, and define the semantics of the language.

Creek is a DSL in Elixir; a functional actor language that runs on the Erlang Virtual Machine (BEAM). Creek has been designed as a concurrent double-phase DSL with push-based semantics, and no backpressure. A double-phase DSL is the evident choice because – as opposed to single-phase – it allows modularity and composition of "partial streams". Because Creek has been implemented on an actor-based language, concurrency was nearly free. We chose to avoid complexity in the base level of Creek by implementing push-semantics. Push-semantics do not have the communication overhead introduced by pull-semantics or backpressure.

2.2.1 Hello, Creek!

Before diving into the details of the semantics of Creek, consider an example Creek application. Listing 4 offers a first taste of the Creek stream DSL. The program defines a stream that produces a list of the squares of the first 100 even numbers.

DAGs For an overview of the syntax of Elixir we refer the reader to appendix A. Every Creek program is defined in an Elixir module. An Elixir module is defined with the `defmodule` syntax (line 1). Creek has a primitive to define Directed Acyclic Graphs (DAGs), `defdag`. The `Evens` module in listing 4 defines three DAGs: `evens` (line 2), `squares` (line 5), and `square_of_evens` (line 7). The `evens` DAG consumes integers from its upstream and filters out all odd

```
defmodule Evens do
    defdag evens as filter(&even?/1)
                    ~> take(100)

    defdag squares as map(&(&1 * &1))

    defdag square_of_evens(input, output) do
        input
        ~> evens
        ~> squares
        ~> output
    end

    def program() do
        src = Source.range(1, :infinity)
        snk = Sink.to_list(self())

        deploy(square_of_evens, [input: src, output: snk])

        receive do
            result -> IO.puts "Squares: #{result}"
        end
    end
end
```

Listing 4: *A first look at Creek: Generating a list of the first 100 squares of even numbers.*

integers. The first 100 even integers are taken from the stream using the take operator. The squares DAG computes the square of each of its inputs. The square_of_evens composes them to create a DAG to compute the squares of the first 100 even numbers. The square_of_evens DAG uses the extended syntax to define a DAG, which allows named parameters to represent the unknown sources and sinks in a DAG definition.

Deploying Streams A DAG is a declarative definition of transformations on an undefined stream of data. A DAG that is paired with a source and a sink, and is processing data, is called a *stream*. Pairing a DAG with sources and sinks is called *deploying* the DAG. A source provides data to the stream, and a sink consumes data from the stream. A stream can have one or more streams and sinks. The sources and sinks of a stream are the connection between the host language and the stream DSL. In context of Creek, an Elixir-hosted DSL, sources and sinks are represented by Elixir actors. In context of RxJava sources and sinks are represented by objects.

In listing 4, a range source is created (line 15); a source that emits all values in a given interval. A to_list sink (line 16) consumes all the values from the stream and sends them as a message to a regular Elixir actor. An instance of the square_of_evens dag is deployed with the source and sink on line 18. Finally, the program awaits the message containing the list of numbers that percolated through the entire stream (lines 20–22) using idiomatic Elixir syntax.

2.2.2 Operators

The fundamental building block of a stream is an *operator*. It is a modular functional unit that is the smallest building block of a stream. Examples of operators are map, filter and zip. All stream DSLs implement a core set of operators, such as map and filter. Table 2.3 lists all the operators that are implemented in Creek.

An operator in a stream is always positioned upstream and downstream of other operators. An operator consumes data from its *upstream* and emits its results to its *downstream*. For example, in listing 4 the take operator on line 3 is the downstream of the filter operator on line 2. Conversely, the filter operator is upstream of the take operator. The take operator has no downstreams, and the filter operator has no upstreams.

Operators are *instantiated* with 0 or more arguments that specify their behavior. Arguments are values from the host language (i.e., Elixir for Creek). In most cases, the arguments are anonymous functions that contain logic that is applied to each datum. E.g., map is instantiated with an anonymous function that the

```
      program  :  [<closedDag>, <openDag>]+, [<deploy>]+

      openDag  :  defdag <name> as <dagExpression>

                  defdag <name>(<name>*) do
    closedDag  :   <dagExpression>
                  end

         name  :  <symbol>

dagExpression  :  <operator>
               |  <dagExpression> ~> <dagExpression>
               |  (<dagExpression> ||| <dagExpression>)
               |  <name>

       deploy  :  deploy(dagExpression, (<symbol: <Pid>)+)
```

Table 2.2: *Creek syntax.*

operator applies to each datum from its upstream. The `from_list` operator is instantiated with a list, of which each value will be emitted downstream by the operator.

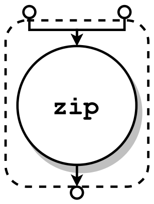


Figure 2.3: *The `zip` operator has two input ports and one output port.*

Every operator has an *arity* that defines exactly how many upstream and downstream operators must be connected to it. An operator has an *input port* for each upstream connection, and an *output port* for each downstream connection. The arity defines exactly how many of each there are. The arity is a property of an operator, and is statically defined. The arity of an operator is expressed as two numbers: `u:d`, where the first number is the amount of *upstream ports*, and the second number is the amount of *downstream ports*. Consider the diagram of the `zip` operator in fig. 2.3. The `zip` operator its arity is `2:1`; it requires

Name	Arity	Args	Description
`map`	`1:1`	Fun	Applies a function to each upstream datum and emits the result downstream.
`filter`	`1:1`	Fun	Applies a predicate to each upstream datum and emits the datum if the predicate returned `true`.
`zip`	`2:1`	/	Merges values from both upstreams into a tuple. Only emits a tuple if both upstreams produced a value. If there is a rate mismatch between upstreams values are buffered. Can cause memory overflow.
`fold`	`1:1`	Fun, Term	Folds over the upstream values by applying the given function to each datum and the accumulator. Emits the accumulator if the upstream is finished.
`dup`	`1:*`	Int	Duplicates values from upstream to all connected downstreams.
`merge`	`*:1`	Int	Merges multiple upstreams into a single downstream. Values are emitted in the order they arrive.
`to_list`	`1:0`	Pid	Accumulates all values from the upstream into a list and sends the list to the given process id (idiomatic Elixir).
`first`	`1:0`	Pid	Waits for a single value from the upstream and sends it to the given process id (idiomatic Elixir).
`last`	`1:0`	Pid	Waits for the upstream to complete and sends the last value of the upstream to the given process id (idiomatic Elixir).
`single`	`0:1`	Term	Emits the given value and completes.
`take`	`1:1`	Int	Takes the first `n` values from its upstream and terminates.
`range`	`0:1`	Int, Int	Produces integers from the given start value to the given end value.

Legend ***Fun***: *An anonymous function,* **/**: *No Arguments,* ***Term***: *Any value,* ***Int***: *Integer,* ***Pid***: *An Elixir Process ID*

Table 2.3: *List of all operators in Creek.*

exactly two upstreams, and exactly one downstream. Some operators, such as the `duplicate` operator, do not have a static number of ports and can be connected to 1 or more times. A *variadic arity* is denoted with an asterisk: `*`. Operators with an input arity of 0 are called source operators, and operators with an output arity of 0 are called sink operators.

Stream DSLs have to interact with the host language. This is done in the paradigm of the host language. Because Creek is a DSL in Elixir, the language makes use of idiomatic Elixir patterns where it interfaces with the host language. Elixir is an actor language, and the means of communication is message-passing between actors. Therefore, source and sink operators need to send messages to communicate with the host application, and to do this they require a *process id* (Pid). For more information on message sending in Elixir, see appendix A.

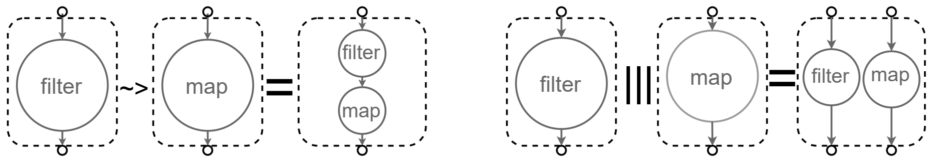
Creek implements a small set of operators which have been chosen based on necessity in developing the evaluation presented in section 4.7. The DSLs based on Reactive Extensions have over 400 operators. However, many of these operators are variations on others, and can be replaced by a combination of two simpler operators. Another large subset of operators are the *higher-order operators*. In Reactive Extensions streams are higher-order values, and the topology of a stream can be changed at runtime. For example, in RxJava the `flatMap` operator transforms a stream of higher-order streams into a single stream by merging the higher-order streams. This process modifies the topology of the stream's DAG at runtime because each higher-order stream is a new source for the downstream. In Creek this is not possible, and therefore Creek has no notion of higher-order operators.

2.2.3 DAGs

The composition of operators into a DAG forms the blueprint of a stream. The order of the composition determines the order of the transformations that will happen to each datum "flowing" through the DAG. An operator itself is the smallest DAG possible, consisting of only the operator itself. DAGs are created by composing DAGs using the *composition functions* `~>` and `|||`.

The vertical composition function (`~>`) connects two DAGs by connecting the output ports of the left DAG to the input ports of the right DAG. When `~>` is applied to left operand a with a_i input ports and a_o output ports and right operand b with b_i input ports and b_o output ports where $a_o = b_i$, the result is a DAG with a_i input ports and b_o output ports. For example, the vertical composition of two DAGs with one input and one output port each results in a DAG that has one input port and one output port. Figure 2.4a shows the vertical composition of `filter` and `map` operator. The result is a DAG with one input port, and one output port.

The horizontal composition function (`|||`) composes two DAGs in parallel. In a horizontally composed DAG, no new connections are made between input- and output ports. When `|||` is applied to DAGs a and b with input and output ports as denoted above, the result is a DAG with $a_i + b_i$ input ports and $a_o + b_o$ output ports. Figure 2.4b shows the horizontal composition of two DAGs with 1 input and 1 output port each, resulting in a DAG with 2 input ports and 2 output ports.



(a) *Vertical Composition.* **(b)** *Horizontal Composition.*

Figure 2.4: *Graphical depiction of the semantics of vertical composition (left), and horizontal composition (right).*

In Creek, there are two types of DAGs. *Open DAGs* and *closed DAGs*. Composing

DAGs results in DAGs that have unconnected input ports or output ports. In Creek, a DAG that has unconnected ports is called an *open DAG*. A DAG is closed when all its input ports and output ports are connected to another operator.

Open DAGs

An open DAG is a DAG that has input- or output ports that are *unconnected*. An in- or output port is unconnected if it has no up- or downstream operator connected to it, respectively. The smallest open DAG that can be created is a single operator. For instance, the expression `map(f)` creates an open DAG with a single operator. An open DAG is created using the syntax shown in table 2.2.

Listing 5 shows an example of an open DAG in Creek. The DAG consists of two operators: a `filter` operator followed by a `map` operator. The only output port of the `filter` operator is connected to the only input port of the `map` operator. The input port of the `filter` operator and the output port of the `map` operator remain unconnected. The DAG is intended to process error events. First, all events except error events are dropped, and then the events are printed to the console, and passed downstream. Because the upstream and downstream of this DAG are undefined, it can be used in the composition of a larger DAG.

```
defdag error_logger as
    filter(fn event -> event.type == :error end)
    ~> map(fn event -> IO.puts "Error: #{event.message}" ; event end)
end
```

Listing 5: *An example of an open DAG. The DAG expects log messages and only prints out the error messages.*

Closed DAGs

A closed DAG is a DAG in which all in- and output ports are connected. The smallest closed DAG that can be created in Creek is a source directly connected to a sink. Listing 6 shows a closed DAG to calculate the product of a stream. The `transform` operator has an arity of `1:1`, and it is connected to both the source and sink. There are no unconnected in- or output ports left in the DAG.

The syntax to create a closed DAG allows introducing variable names between parentheses after the DAG name.

These variables are called *actor sockets* and can be used as sources or sinks in the scope of the DAG definition. Actor sockets are placeholders for sources and

```
defdag product(src, snk) do
    src
    ~> transform(1, fn v, acc -> {acc * v, acc * v} end)
    ~> snk
end
```

Listing 6: *A DAG to compute the product of a stream.*

sinks, and are replaced by Elixir actors when the DAG is deployed. An actor socket can be used in a position that requires an in- or output arity of 1 and an out- and input arity of 0, respectively. Consider the example in listing 7. The `log_print` DAG introduces two actor sockets: `log_src` and `log_snk`. The `log_src` actor socket is placed in source position and serves as a placeholder for a source actor. The `log_snk` actor socket is placed in sink position and serves as a placeholder for the sink actor. In the example the source is connected to the `error_log` DAG (listing 5), and the output of the `error_log` DAG is connected to the sink. The `error_logger` DAG has an arity of `1:1`, meaning `log_src` is indeed in a `0:1` position and `log_snk` in a `0:1` position.

```
defdag log_print(log_src, log_snk) do
  log_src
  ~> error_logger
  ~> log_snk
end
```

Listing 7: *An example of a closed DAG. The actor sockets `log_src` and `log_snk` will be replaced with actors once the DAG is deployed.*

Valid DAGs In Creek, any closed DAG that is deployed must adhere to a set of well-formedness constraints. If all these constraints are met the DAG is called a *valid dag*, and can be deployed.

1. Each output port is connected to an operator's or sink's input port and each input port is connected to an operator's or source's output port.
2. Operators have precisely as many connections as ports.
3. Actor sockets have exactly one in- or output port connected.
4. No cycles are created in the DAG.

2.2.4 Streams

A DAG is a *description* of a stream but does not process any data, which is why, in Creek, it is informally called "a blueprint for a stream". A closed and valid

DAG in a Creek program can be turned into a stream by *deploying* it.

To deploy a DAG, plain Elixir actors need to be provided to replace the actor sockets in the DAG definition (`log_src` and `log_snk` in listing 7). Creek is a DSL in Elixir, and therefore the sources and sinks are actors. To deploy the `log_print` DAG, a source actor and sink actor need to be provided. The source actor provides the data, and the sink actor consumes the transformed data. Listing 8 shows how the DAG defined in listing 7 can be deployed. Any Elixir actor can act as a source and a sink provided that they implement the protocol for communicating with a stream. In Elixir, this means that the actors must understand a specific sequence of messages. We go into detail of this protocol in the next section.

When a Creek DAG is deployed, each operator in the DAG is turned into a concurrent entity. An operator in a stream is either an operator (e.g., `map`), a source, or a sink. Once a DAG has been deployed, it runs asynchronously from the actor that started it. The result of a deployment is an immutable reference to a stream, that can be used to determine if a stream is still alive or not.

```
source_actor = ...
sink_actor = ...
stream = deploy(log_print, log_src: source_actor, log_snk: sink_actor)
```

Listing 8: *Example of deploying a DAG in Creek.*

2.2.5 The Canonical Stream Protocol

To propagate a datum through stream, internal messages are sent between the operators of that stream. In Creek, the protocol that defines the events an operator must understand and is allowed to emit is called the *Canonical Stream Protocol (CSP)*. The CSP defines the structure and order of the messages an operator can send to its up- and downstreams, or receive from its up- and downstream. The CSP defines the interactions for each type of operator: sources, sinks, and operators. The operator implements different parts of the protocol, depending on its type (i.e., source, sink, or operator). In this section we introduce the protocol and define its limitations and guarantees.

Operator Canonical Stream Protocol

Figure 2.5 shows the CSP for a single operator (i.e., not a source or sink) in a stream (e.g., `map` or `filter`). An operator can receive events from its upstream and downstream, denoted as `rcv e` in the case of an incoming event *e*. If an operator emits a message, it is denoted as `snd e` where *e* is the event. The

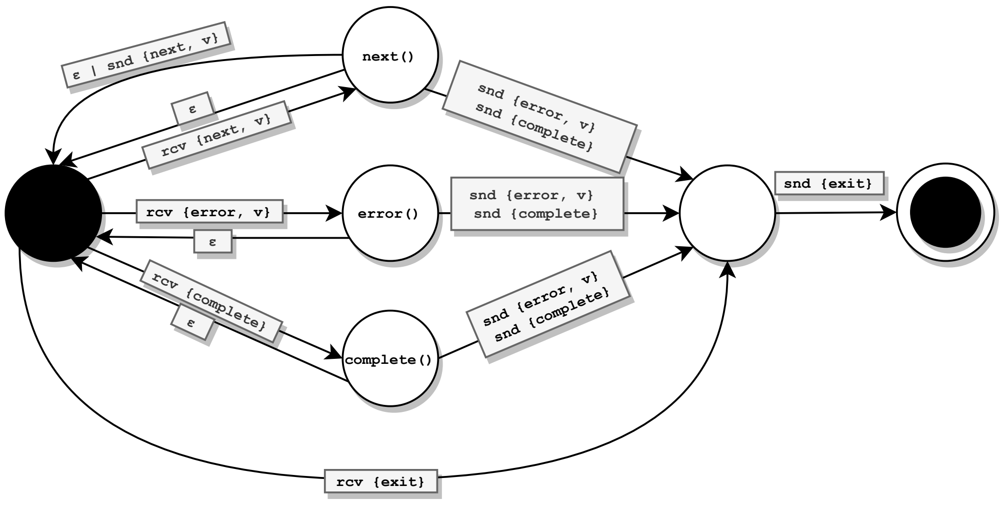


Figure 2.5: *Canonical Stream Protocol for operators. The events are denoted between curly braces and prefixed with whether they are sent (`snt`) or received (`rcv`).*

diagram represents a communication protocol, and each vertex represents a state from which a set of valid messages can be either received or sent. The black vertex is the initial state after a stream has been deployed. The black vertex with a circle is the final state. When an operator arrives in the final state, it can no longer send or receive messages. The operator is said to be *terminated*. When no messages are sent, the ϵ transition is denoted.

The CSP for an operator defines four messages that an operator must understand.

- `{next, v}` is an event sent by an operator connected to an input port of the receiving operator. The `next` event carries an application-level datum `v`.
- `{error, v}` is an event sent by an operator connected to an input port of the receiving operator. It signals an error occurred in the upstream. The error is contained in the value `v`.
- `{complete}` is an event sent by an operator connected to an input port of the receiving operator. It signals that the upstream has completed and will no longer emit new data.
- `{exit}` is an event sent by an operator connected to an output port of the receiving operator. It signals that the downstream operator has completed and no longer accepts events.

The CSP for an operator defines four types of messages that an operator is allowed to send.

- `{next, v}` is an event that an operator can send to the operators connected to its output ports. The message contains a new application-level datum `v`.
- `{error, v}` is an event that an operator can send to the operators connected to its output ports. The message contains an error value `v`, signaling an error occurred somewhere in the stream. An error can also be sent in response to a `next` message, if the error occurs when processing a `next` event.
- `{complete}` is an event that an operator can send to the operators connected to its output ports. It signals that the operator has completed and will no longer emit new data.
- `{exit}` is an event that the runtime can send to the operators connected to its input ports. It signals that the operator has completed and no longer accepts events. The `{exit}` message is automatically sent when an operator propagates an error or a completion downstream.

The logic that handles events is called a *behavior*, similar to a method in object-oriented programming. The result of executing a behavior is an *instruction* to the runtime. Based on this instruction, the runtime sends the appropriate messages to the down- and upstream operators. All the incoming events, except the `exit` event, must be handled by the operator.

Listing 9 shows an example implementation of the operator protocol. The behavior defines the `filter` operator. Defining an operator consists of creating three functions that handle the three different incoming messages. There are three behaviors defined, i.e., `next`, `error`, and `complete`. These three functions constitute the behavior of the operator. When a `next` event arrives, the `filter` predicate is applied, and if it returns `true` the instruction to emit a `next` event is returned with the incoming datum as payload. If the predicate returns `false` the value is skipped by returning the `skip` instruction (`skip` is represented as ϵ in fig. 2.5), telling the runtime to not send any messages downstream. When the operator receives a `complete` event, indicating the upstream will no longer produce events, the event is passed downstream.

Source Canonical Stream Protocol

Contrary to operators, sources generate values without ever receiving a value. The `tick` event is a cue to the source operator that it should produce a single value, generated by the Creek runtime. Consequently, the only behavior that

```
defmodule Operator.Filter do
  def next(this, state, value) do
    if this.arg.(value) do
      {state, :next, value}
    else
      {state, :skip}
    end
  end

  def error(this, err) do
    {state, :error, err}
  end

  def complete(this, state) do
    {state, :complete}
  end
end
```

Listing 9: *Implementation of the `filter` operator in Creek.*

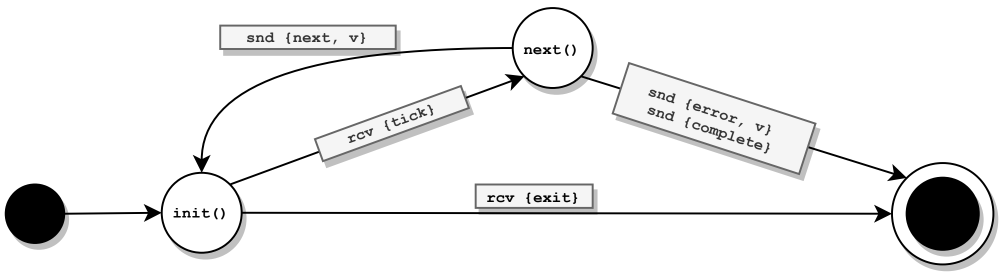


Figure 2.6: *Canonical Stream Protocol for source operators. The events are denoted between curly braces and prefixed with whether they are sent or received.*

needs to be implemented in a source is the `tick` behavior. The CSP for sources defines two types of incoming messages an actor must understand.

In Creek, a source operator is a plain Elixir actor that implements the CSP. It must understand the following messages.

- `{exit}` is an event that an operator can send to the operators connected to its input ports. It signals that the operator has completed and no longer accepts events.
- `{tick}` is an event that the runtime sends to all the sources in a stream. It signals that the source must produce a new datum.
- `{init}` is an event that the runtime sends to all the sources in a stream when the stream is deployed. It allows the source to execute initialization logic (e.g., database connections).

As an example implementation of a source behavior, listing 10 contains the implementation of the `range` operator. The `range` operator is instantiated with three arguments: the start and end value of the interval and its stepsize. For each `tick` event, the source returns the start value of the interval, and updates its interval. If all values in the interval have been emitted, the source completes.

```elixir
defmodule Source.Range do
  def tick(this, state) do
    {a, b, s} = state

    if b == :infinity or a <= b do
      {{a + s, b, s}, :next, a}
    else
      {state, :complete}
    end
  end
end
```

Listing 10: *The implementation of the `range` operator in Creek. Every time the `tick` event is received from the runtime, the range operator generates the next value in the interval. When all values have been emitted, the range operator completes.*

Sink Canonical Stream Protocol

In Creek, a sink operator is a plain Elixir actor that respects the CSP for sink actors. The CSP for a sink has overlap with the CSP for operators, and here we only highlight the differences. Figure 2.7 shows the CSP for a sink operator. Notice that the sink cannot receive an `{exit}` message from an operator connected

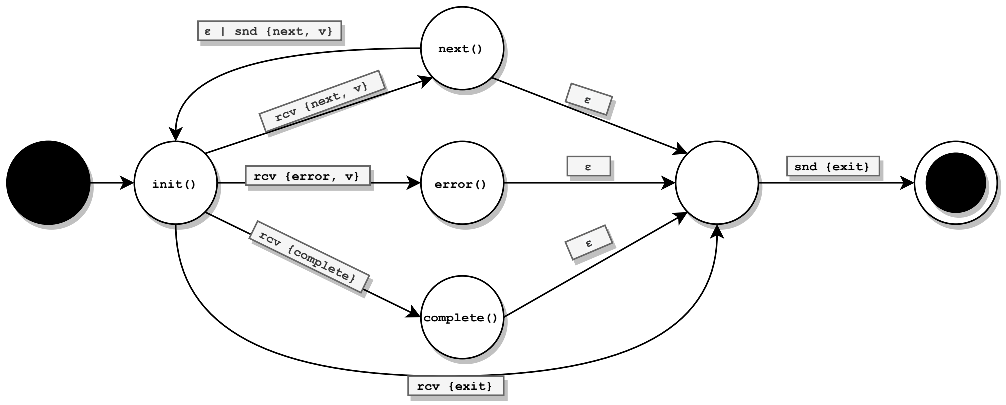


Figure 2.7: *Canonical Stream Protocol for sink actors. The events are denoted between curly braces and prefixed with whether they are sent or received.*

to its output ports, because a sink does not have any output ports.

The CSP for sinks defines the only message a sink operator can send, the `{exit}` message. Given that a sink has no downstream, it can only communicate with its upstream, and the only message that is sent upstream is the `{exit}` message.

A sink operator does not send messages to downstream operators. Therefore all transitions are labeled as ϵ.

As an example of a sink operator implementation, consider listing 11. It shows the source code for the `to_list` operator in Creek. The `to_list` operator accumulates all the values it receives, and when it has completed – either normally or by due to an error – it will send the list of values to the given Elixir actor. The arguments to create a `to_list` sink are a process identifier of another actor that should receive the list of values. For every `next` event, the sink accumulates it in a list and waits for the next event. If an error is encountered in the upstream, the values aggregated up to that point and the error value are sent to the argument actor. In case the upstream terminates with a `complete` message, the whole list is sent to the actor.

2.3 Stream Termination Protocol

In the previous sections, we introduced the protocol used to propagate data through a stream. The application-level data is propagated by means of `{next, v}` messages. The `error` and `complete` messages signal to the rest of the stream that an operator terminated. The `exit` message is emitted upstream, and signals

```elixir
defmodule Sink.ToList do
  def next(this, state, value) do
    {receiver, xs} = state
    xs = [value | xs]
    {{receiver, xs}, :ok}
  end

  def error(_this, state, err) do
    {receiver, xs} = state
    send(receiver, {:error, err, xs})
    {state, :error}
  end

  def complete(_this, state) do
    {receiver, xs} = state
    send(receiver, {:ok, xs})
    {state, :complete}
  end
end
```

Listing 11: *Implementation of the `to_list` operator in Creek.*

that a downstream operator has terminated. This can lead to situations where a source is producing data, but the downstream is only partially available. In such a stream it is impossible for data to propagate. We call such a stream a *useless stream*. The stream termination protocol is designed to terminate all operators in a stream when it becomes useless.

Consider the following scenarios that sketch a proof that the CSP avoids useless streams.

A stream consists of two operators A and B with arity `1:1` where A is the upstream of B. If we assume that A terminates, there can be two causes. Or, 1) A received an `exit` message from B, or 2) it emitted `{complete}` or `{error, v}` to B. In the first case, it came from B, and the CSP tells us that B is therefore terminating. In the second case the CSP dictates that B will terminate as well, and A will emit an `exit` event to its upstream to propagate the termination in both directions. In this scenario, the termination will propagate both upstream and downstream, terminating the entire stream.

In case an operator has an upstream arity larger than 1, its behavior must decide what to do in case only one of its upstream completes. This behavior depends on the operator. An operator can signal it wants to continue operating until it has no upstreams left, or terminate as soon as one of its operators terminates. For example, a `zip` operator can not function without both of its upstreams, while a `merge` operator can function with only one of its upstreams.

We conclude that the CSP ensures that a stream will terminate as soon as it has become impossible for data the propagate through the stream. However, the CSP will not terminate an entire stream if there are still paths of the stream through which data can propagate.

2.4 Summary of Chapter

This chapter introduced the concepts involving stream Domain Specific Languages (DSLs). We explained streams, Directed Acyclic Graphs (DAGs), operators, and stream deployment.

We surveyed the state of the art and taxonomized the DSLs along four design principles: execution semantics, phases, propagation semantics, and backpressure. We concluded that all DSLs except Akka Streams are single-phase DSLs and that all DSLs implement pull-based semantics.

Based on the design principles, we designed a DSL called Creek with push-based semantics, two phases, no backpressure, and concurrent execution semantics.

We described the canonical stream protocol, which defines the messages between operators in a stream in Creek. We conclude by defining the stream termination protocol, which ensures that failures in a stream do not leave the software in an inconsistent state.

In the next chapter, we use Creek as a basis to formulate a stream-based meta-level architecture called μCreek.

Chapter 3

μCreek: A Meta-Programming Approach for Stream Languages

In the previous section we introduced Creek, a new prototypical stream Domain Specific Language (DSL) that is embedded in Elixir. In chapter 4 Creek will be used as the stream language in a Cyber-Physical System (CPS) framework. Chapter 1 discussed the challenges in designing CPSs. One of the challenges was dealing with the heterogeneous nature of the devices. This heterogeneity introduces differences in communication patterns, data encoding, performance requirements, and reliability. We call these concerns *non-functional concerns* of a stream-based program. In this section we add a meta-level architecture to Creek to separate these non-functional concerns from the application logic. First we introduce our motivation to do so, then we explain the approach technically, and conclude with a performance evaluation.

3.1 Motivation

Stream programming is useful for expressing computations that are naturally specified as a pipeline of transformations, such as data flux between devices. However, the state of the art in stream Domain Specific Languages (DSLs) is not always able to express the non-functional aspects of an application in a satisfactory way. In a Cyber-Physical System (CPS), these non-functional concerns are introduced by the heterogeneous devices, and non-functional requirements of the application. Hence the motivation to add new features to stream-based languages to cater to these non-functional concerns. The "Separation of Concerns" (SOC) principle dictates that functional and non-functional concerns should be separated.

Contemporary and widely used stream DSLs such as Akka Streams [66], RxJava [82], and Flow [3] offer a high-level declarative API to construct and manipulate streams of data by using functional-inspired building blocks called "operators", such as `map` and `filter`. These basic building blocks can be classified into two categories, which are central to our problem statement:

- *Functional operators* such as `map`, `filter`, and `scan`, that manipulate the data flowing through streams.
- *Non-functional operators* that manipulate *how* the data streams through the stream such as `buffer` and `async`.

The API for expressing the *functional* requirements of a stream program are homogeneous between the different DSLs. This is not the case, however, when it comes to expressing the *non-functional* requirements of stream programs. Surveying six existing stream DSLs embedded in popular general-purpose languages [4](PLINQ in C#, Streamz in Python, Akka Streams in Scala, Java Streams and RxJava in Java, and Flow in Elixir), we identified three problems with respect to expressing non-functional requirements in these DSLs. These types of issues have been addressed in the past in the context of aspect-oriented programming [59, 60] and meta-programming [20, 15, 42] and we wish to address a subset of them in our work.

1. Stream DSLs provide non-functional extensions in a non-canonical and *ad-hoc* fashion, hindering transplantability of programs between languages.
2. *Entanglement* of functional stream logic and non-functional stream execution logic, leading to reduced readability and maintainability of stream programs.
3. Limited or *no means for extending and adapting the stream execution semantics* from within the DSL beyond what is built into language.

3.2 Problem Statement

In section 3.1, we have presented three problems in the state of the art of stream DSLs in contemporary programming languages. In this section we substantiate these three claims. More specifically, we show by example that contemporary stream DSLs provide non-functional extensions in a non-canonical and ad-hoc fashion (section 3.2.1), that they entangle functional and non-functional stream logic (section 3.2.2), and – most importantly – that there are limited or no means for extending and adapting the stream execution semantics from within the DSL beyond what is offered by the language (section 3.2.3).

Although we will discuss each problem in the context of only one or two of the prominent streaming libraries we surveyed for this purpose, we observed all languages suffer from at least one or more of the posited problems.

3.2.1 Problem #1: Lack of Canonical Non-Functional Operators

Stream execution semantics is concerned with how the definition of a stream is executed at run-time to reduce the stream to a result. Given the wide range of applications in which stream DSLs are applicable, there are situations where the default stream execution semantics offered by a stream DSL does not (or no longer) match with the intended semantics. Stream DSLs cater to this problem by offering non-functional operators to manipulate how a stream is executed with respect to propagation semantics (push, pull, backpressure, ...), concurrency and parallelism, buffering, error handling, etc. These operators are used to improve on one or more non-functional requirements such as scalability, performance, monitoring, maintainability, or recoverability. However, while the typical building blocks for expressing the functionality of stream programs are highly similar across different stream DSLs, this is not the case for the non-functional operators. Moreover, different DSLs offer different sets of non-functional operators to support different non-functional requirements. In brief, there is no *canonical* set of non-functional operators.

Example: Error Handling in Akka Streams and RxJava

```scala
val decider: Supervision.Decider = {
  case _: IllegalArgumentException => Supervision.Restart
  case _                           => Supervision.Stop
}
val flow = Flow[Int]
  .scan(0) { (acc, elem) =>
    if (elem < 0) throw new
                    IllegalArgumentException("negative not allowed")
    else acc + elem
  }
  .withAttributes(ActorAttributes.supervisionStrategy(decider))
val source = Source(List(1, 3, -1, 5, 7)).via(flow)
val result = source.runWith(Sink.seq)
```

Listing 12: *Error handling in Akka Streams using* `Decider`. *All* `IllegalArgumentException`*s cause a restart, and other exceptions stop the stream execution.*

Stream DSLs offer facilities for the programmer to deal with exceptions during stream execution. For instance, Akka Streams modularizes exception handling of a stream by extracting the exceptional behavior into a separate object called a `Decider`. The programmer defines what should happen in case of a specific exception class. The example in listing 12 defines a `Decider` object that, when an `IllegalArgumentException` occurs, restarts the stream entirely. If the stream produces any other type of error the entire stream is aborted and is reduced to an error value. If the source of the stream produces the erroneous value again the stream will restart, possibly creating an infinite loop if no limit is placed on the restarting.

Decider behavior can be added in a modular fashion, separated from the actual stream logic. For instance, the stream program in listing 12 instructs the `scan` operator to employ the `decider` object. However, not all stream DSLs offer this kind of (modular) error handling. In RxJava, for example, one cannot express behavior equivalent to the `Decider` object. The only way for the programmer to handle errors is by changing the definition of the stream by inserting additional non-functional operators such as `retry` and `retryWithValue`. We do note that any approach discussed here re-executes the entire stream. Ignoring an erroneous value is impossible.

Example: Ad-hoc Parallelization in RxJava and Akka Streams

Suppose that a stream program has to map some long-lasting computation (represented by the function `longLastingComputation`) over a stream of values, and that applying this operator in parallel would increase throughput. Listing 13 shows a small RxJava program containing the `longLasting⌋ Computation`. The left-hand side is the sequential version, and the right-hand side is a parallelized version. Due to the poor integration of parallelism in RxJava, the stream programmer is forced to modify the layout of the stream substantially to facilitate for the parallelism (lines 3–7), making its functionality less readable. Listing 14 contains the same sequential example program on the left, and the parallelized version on the right, but written in Akka Stream. As in RxJava, parallelizing a single operator introduces a significant amount of non-essential complexity, significantly increasing the number of LoC. From the code in listings 13 and 14 we make the following observations:

- Although both RxJava and Akka Streams allow the parallelization of operators, *the mechanism to do so is very different.* RxJava requires the programmer to manually place computations on different threads of computation, while Akka Stream requires the programmer to manually create parallel pipelines of computation.
- Although parallelization is a non-functional concern, programmers in RxJava and Akka Streams have to *modify the original stream to express this requirement*.

```
source
.map(i ->
  longLastingComputation(i))
.subscribe()
```

```
source
.subscribeOn(Schedulers.computation())
.flatMap(val ->
    just(val)
    .subscribeOn(Schedulers.computation())
    .map(i -> longLastingComputation(i))
    .subscribeOn(Schedulers.single()))
.subscribe();
```

Listing 13: *An RxJava Stream program that applies* `longLastingComputation` *over each datum in the stream. The sequential program is shown on the left, and a parallelized version on the right. The parallelization factor is hidden in the* `Schedulers` *definition.*

```
source
.map((i) =>
  longLastingComputation(i))
.runForEach(println)
```

```
val processor: Flow[Int, Int, NotUsed] =
  Flow.fromGraph(GraphDSL.create() { implicit b =>
    val balance = b.add(Balance[Int](2))
    val merge = b.add(Merge[Int](2))
    val f = Flow[Int].map(longLastingComputation)

    balance.out(0) ~> f.async ~> merge.in(0)
    balance.out(1) ~> f.async ~> merge.in(1)
    FlowShape(balance.in, merge.out)
})
source.via(processor).runForeach(println)
```

Listing 14: *An Akka Stream program that applies* `longLastingComputation` *over each datum in the stream. The sequential program is shown on the left, and a parallelized version that uses a parallelization factor of 2 is shown on the right.*

Conclusion #1

Stream DSLs feature non-functional operators that address non-functional concerns. However, opposed to the work in Object-Oriented Programming [59, 60], **there is no canonical approach to deal with non-functional concerns**. Stream DSLs approach non-functional concerns in an ad-hoc fashion, resulting in completely different approaches for the same problem in different DSLs.

3.2.2 Problem #2: Entanglement of Functional and Non-functional Operators

In stream programming, the domain logic of the stream (i.e., *what* the stream computes) is a separate concern from the execution semantics of the stream (i.e., *how* the stream executes). In section 3.2.1 we argued that there is no uniformity or idiom present in the existing non-functional operators. In this section, we argue that there is a need to separate non-functional operators from the functional operators.

The software engineering principle of “Separation of Concerns” (SoC) [36] dictates that the different concerns of an application should be defined in different parts of the application, separated by a clean interface. SoC implies that functional and non-functional operators should not be mixed in the definition of a stream. Additionally, using non-functional operators in stream DSLs often forces the definition of the stream to be modified to suit the inclusion of those operators. In section 3.2.1 we have already encountered examples where this was the case; in the larger example that follows, we revisit parallelization in Akka Streams.

Consider the example programs in listings 15 and 16, that analyze a stream of tweets containing the keywords “covid19” and “sars-covid” for its sentiment: positive, neutral, or negative. Every 100 tweets, the programs print the ratio of positive, negative, and neutral sentiments to the console. An example output is `(5, 25, 70)`, indicating that in the last 100 tweets, 5%, 25%, and 70% of all tweets were positive, neutral, and negative respectively.

The program in listing 15 defines the domain logic to achieve this in Akka Streams. Each tweet emitted by the actor is transformed into a tuple of sentiments (lines 3–7). Finally, the amount of positive, neutral, and negative sentiments is printed to the console.

Given that sentiment analysis is a computationally expensive computation, it might be opportune to parallelize that part in the stream to improve the throughput of the entire stream. Parallelizing the application from listing 15 requires two major changes to the stream definition. First, the sentiment analysis

```scala
val source = TwitterActor.mkActor()
val actorRef = source
    .flatMapConcat((tweet: Status) => {
        var result: = Sentiment.computeSentiment(tweet.getText())
        Source(result)
    })
    .map(tuple => tuple._2)
    .sliding(100)
    .map((win: Seq[Sentiment]) => {
        (win.count(s => s == Sentiment.POSITIVE),
        win.count(s => s == Sentiment.NEUTRAL),
        win.count(s => s == Sentiment.NEGATIVE))
    })
    .toMat(Sink.foreach(println))(Keep.left).run()

TwitterActor.pipeInto(Array("covid19", "sars-covid"), actorRef)
```

Listing 15: *Computing the average sentiment of tweets related to COVID-19 sequentially in Akka Streams.*

```scala
val analyze = Flow[Status].flatMapConcat((tweet: Status) => {
    var result  = Sentiment.computeSentiment(tweet.getText())
    Source(result)
  })
val analyzer = Flow.fromGraph(GraphDSL.create() { implicit builder =>
    val dispatchTweets = builder.add(Balance[Status](2))
    val mergeSentiments = builder.add(Merge[(String, Sentiment)](2))
    dispatchTweets.out(0) ~> analyze.async ~> mergeSentiments.in(0)
    dispatchTweets.out(1) ~> analyze.async ~> mergeSentiments.in(1)
    FlowShape(dispatchTweets.in, mergeSentiments.out)
  })
val source = TwitterActor.mkActor()
val actorRef = source
    .via(analyzer)
    .map(tuple => tuple._2)
    .sliding(100)
    .map((win: Seq[Sentiment]) => {
      (win.count(s => s == Sentiment.POSITIVE),
       win.count(s => s == Sentiment.NEUTRAL),
       win.count(s => s == Sentiment.NEGATIVE))
    })
    .async
    .toMat(Sink.foreach(println))(Keep.left).deploy()
TwitterActor.pipeInto(Array("covid19", "sars-covid"), actorRef)
```

Listing 16: *Computing the average sentiment of tweets related to COVID-19 in parallel in Akka Streams.*

step must be factored out into a separate reusable `flow` (lines 1–3 in listing 16) so that it can be used twice in a parallel pipeline. Second, a parallel pipeline needs to be created that balances the tweets over two sentiment analyzers (lines 5–10 in listing 16) to avoid processing the same tweet twice. Listing 16 requires four operators to facilitate the parallelism: a `Balance` operator to spread the tweets over two sentiment analysis steps (line 6), two sentiment analysis operators (lines 8, 9), and a `Merge` operator (line 7) to merge the results from both sentiment analysis back together. In brief, to parallelize one operator, three additional operators are required in the stream definition.

Conclusion #2

The example of parallelizing tweet sentiment analysis using Akka Stream shows a problem that is present in many stream DSLs. **For a stream definition to exhibit specific non-functional properties – parallelism in this case – the domain logic needs to be adapted** causing it to become less readable, more complex, and less adaptable to changing non-functional concerns.

3.2.3 Problem #3: Hard-coded Execution Semantics

Every Stream DSL offers an ad-hoc set of non-functional operators for expressing different non-functional requirements. Because stream DSLs are applicable in many different situations, the intended semantics of the language does not always match the language‘s semantics. None of the stream DSLs we surveyed, however, provide a structured means for extending and adapting the stream execution semantics from within the DSL itself. All the stream DSLs we investigated implement either *push-based* or *pull-based* propagation by default, or a variation thereof [32].

- **Producer-Driven Applications** Push-based semantics is a good fit for *producer-driven* applications in which, every time the producer produces a new datum, the stream computes its result. In this semantics, the producer dictates the rate at which the stream computes.
- **Consumer-Driven Applications** Some applications are inherently *consumer-driven*: the consumer demands data when it is needed downstream. For consumer-driven applications *pull-based* propagation is a better fit than push-based propagation: the consumer will “pull” data as is needed, meaning that the consumer dictates the rate of the stream.

We illustrate this problem in Akka Streams. Akka Streams uses a complex backpressure algorithm [32] that ensures that production and consumption rate in the stream converge to equilibrium. However, in producer-driven scenarios, the programmer wants pure push-based semantics. Consider the example of thermometers monitoring a thermal processing pipeline. For safety reasons, it may be necessary that the thermometers measure as frequently as possible, meaning that it is a producer-driven stream. Consider the code in listing 17 that contains an example program that processes a remote stream of measurements from a thermometer.

```
remoteThermometer
    .buffer(100, OverflowStrategy.dropHead)
    .map(f => logToDB(f))
    .map(f => celsiusToFahrenheit(f))
    .runWith(foreach(f => {
        printf("Current temp: %fF\n", f))})
```

Listing 17: *Processing temperature measurements with a `buffer` operator to simulate pull-based semantics in Akka Streams.*

The programmer intended to write a producer-driven stream to ensure that the sensor measures as frequently as possible. But, due to the default semantics of the stream library (i.e., backpressured pull-based) this is not possible, and the sensor will only measure at a frequency determined by the speed at which the stream can process a value. The stream will run at an equilibrium between the highest rate the thermometer can measure and the rate the system can process the measurements if the `buffer` operator (highlighted in listing 17) is omitted. By using backpressure the thermometer measures less frequently, but no data is lost. The intended semantics could have been to measure as frequently as possible, at the cost of losing some data.

To emulate the intended semantics, a non-functional operator such as `buffer` can be used. The `buffer` operator accepts data at the rate the upstream produces, and buffers it for the slower downstream to consume, consuming values at unbounded speed. This introduces a potential memory leak, so the buffer operator in the case of Akka Stream, is instantiated with an upper limit on the buffer size, and an action to take if the buffer overflows. In listing 17 the buffer will hold 100 values, and drop the oldest value in the buffer to make room for new values.

In summary, the `buffer` operator ensures that the rate between the source and the `buffer` operator will be the maximum rate of the source. The rate between the `buffer` and rest of the stream is dictated by the slowest link. If the source continuously produces at a faster rate, however, data loss is unavoidable.

Conclusion #3

Propagation semantics are hard-coded into the stream DSL implementations. The programmer can only manipulate the streams in a limited fashion by using non-functional operators such as `buffer`. The downsides are that the domain logic becomes polluted with non-functional operators. Second, the intricacies of the underlying implementation of the propagation semantics needs to be well understood to use these operators and place them in the right position in the stream, making the solution very application-specific. With the current state of affairs, **there is no structured canonical approach to modify the propagation semantics.**

3.3 Meta-Programming: Terminology

Before we discuss our approach, we introduce common terminology in the context of meta-programming. Meta-programming has evolved over the past 40 years, starting with Friedman and Wand[42] and Smith[88]. We do not wish to give a historical account of meta-programming, however, we do present an overview of the work that is relevant to this dissertation.

Any computational system (i.e., a program) is designed to reason about a specific domain. A computational system represents its domain with internal data structures. A system that can reason about itself, and possibly modify itself, is called a *reflective system* [68]. When a reflective system reasons about itself, the process is called *introspection* [20]. When the system modifies itself by means of reflection, the process is called *self-modification* [20]. When the system modifies not the system itself, but the semantics of the evaluator of the system (i.e., the semantics of the programming language), the process is called *intercession* [20]. When a reflective system modifies the structure of the program, the process is called *structural reflection* [20]. When a reflective system modifies the behavior of a program, the process is called *behavioral reflection*. Macros are a well-known example of structural reflection, aspects are an example of behavioral reflection.

A language is said to have a *reflective architecture* [68] if reflection is a fundamental part of its design. The reflective architecture is the set of abstractions and mechanisms that facilitate reflection. Reflective systems are programmed against the reflective architecture (i.e., API). The reflective systems' computations are executed at the *meta-level*. The internal data structures from the system that are represented at the meta-level are said to be *reified* [42] representations. The system that is reflected upon is called the *base-level*. Two systems are *causally connected* when changes in one system affect the other system, and vice versa. For example, the digital representation of an airconditioning unit represents the physical device using internal data structures and functions. When changes to these data structures affect the physical device, and vice versa, they are causally connected.

In the past years many approaches have been proposed to create reflective systems. Architectures have been proposed for procedural [88], logic-based [19], rule-based [41], and object-oriented languages [20, 34, 59]. The design of a reflective architecture should reify just enough of the internal data structures to allow the expression of meta-level programs. On the other hand, the reflective architecture should not reify too much internal data structures to avoid introducing unnecessary complexity in the meta-level programs.

In what follows we explain our reflective architecture for stream Domain Specific Languages (DSLs), called μCreek.

3.4 Separating Domain Logic in Streams

We introduced Creek in chapter 2, a stream Domain Specific Language (DSL) built on top the Canonical Stream Protocol (CSP) introduced in section 2.2.5. The protocol defines the messages operators in a stream send to each other to propagate data. In section 3.1 we introduced three problems with contemporary stream DSLs: lack of canonical non-functional operators, entanglement of functional and non-functional operators, and hard-coded execution semantics.

In this section we introduce our approach to tackle these issues in stream DSLs. We introduce μCreek, a meta-level architecture built on top of Creek. μCreek is a two-pronged approach. First of all, μCreek$_R$ is a *run-time meta-level* (i.e., behavioral reflection) for streams that allows streams' behavior to be changed at run-time. Second, μCreek$_C$ is a *compile-time meta-level* (i.e., structural reflection) that can change the structure of Directed Acyclic Graphs (DAGs) before they are deployed as streams. We conclude with examples of solutions to the problems we addressed in section 3.1.

3.4.1 μCreek$_C$: Compile-Time Meta

As part of our two-pronged approach in μCreek, we propose *structural reflection* to solve the mixing of functional and non-functional operators and the fact that non-functional operators are not canonicalized across stream DSLs. Our approach is implemented on top of Creek, and called μCreek$_C$.

In μCreek, each DAG is compiled into an internal representation, as shown in fig. 3.1. *Structural reflection* offers hooks into this process to modify its semantics. We propose *structural reflection* because (i) it is defined separately from the domain logic and it decouples from any specific domain logic by construction, (ii) if the intercession language is expressive enough it can subsume a set of individual (non-)functional operators, and (iii) the concept of intercession is not bound to a specific stream DSL, but rather the API it exposes.

In the remainder of this section we introduce our approach to *structural reflection* in our stream DSL, μCreek$_C$, which allows the meta-level to manipulate the structure of the DAG at compile-time in a stream paradigm. We then introduce the intercession language of μCreek$_C$, and we conclude with an example of applying *structural reflection* for fusing consecutive `map` operators in a stream (i.e., *operation fusion*). Finally, we discuss Creek's design principles (section 3.4.1).

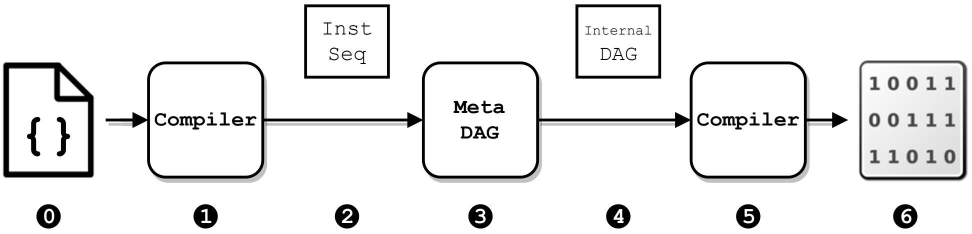


Figure 3.1: *A graphical depiction of the compilation pipeline in μCreek.*

Instruction Stream

To enable structural reflection, the user program has to be *reified*; it needs to be represented in such a way that it is easy to transform at the *meta-level*. The reified representation is a more abstract representation of the internal data structures of the compiler. The reification of the program defines the interface the meta-program uses to express transformations. In μCreek$_C$, each DAG defined in the user program is reified as an *instruction stream* (step 2 in fig. 3.1). An instruction stream is a stream of DAG instructions that – when all executed – create the DAG as it was defined by the programmer. Figures 3.2 and 3.3 show an example of a DAG and its reified representation. Figure 3.2 defines

```
defdag even_square as
    filter(&even?/1)
    ~> map(&Math.square/1)
```

Figure 3.2: *Definition of a simple open DAG that filters out all odd numbers and squares the even numbers.*

```
{:operator, :filter, &even?/1}
{:name_it, "x"}
{:operator, :map, &Math.square/1}
{:name_it, "y"}
{:edge, "x", 0, "y", 0}
```

Figure 3.3: *Instruction stream after compiling fig. 3.2.*

the `even_square` DAG by vertically composing a `filter` and `map` operator. Figure 3.3 defines the reified representation of `even_square`. The first two instructions define an operator of type `filter` with a function as its argument, and label it with the name `"x"`. The next two instructions define a `map` operator with a function as its argument and label it `"y"`. The last instruction connects the first output port of the `filter` operator to the first input port of the `map` operator.

A μCreek$_C$ program takes as input an instruction stream and produces an instruction stream, hence, it is a stream operator. There are **three different types of instructions** that can be processed by the meta-program.

- **Operator Instruction** Each operator in the DAG is reified as an `operator` instruction. The meta-level has access to the type of operator, e.g., `map`, and its arguments, e.g., the function in a `map` operator. The meta-level can change the type of an operator, as well as its arguments. For example, the instruction `{:operator, :map, &Math.square/1}` defines a `map` operator with the `&Math.square/1` as its argument.

- **Edge Instruction** Each connection between two operators is reified as an `edge` instruction. The meta-level has access to the operators and their port indices. The meta-level can change the operators between which the edge is drawn, the in- and output ports of these operators, operating remove an edge. For example, the instruction `{:edge, "x", 0, "y", 1}` in fig. 3.3 draws an edge from the first output port of `"x"` to the second input port of `"y"`.

- **Naming Instruction** Each operator is reified as an operator instruction. These instructions do not refer to an operator in the DAG, and therefore have no referential identity. To give the meta-level a mechanism to reference operators in the DAG, each operator is aliased with a variable name. Each operator instruction is followed by a naming instruction that defines the operator its alias. For example, the instruction `{:name_it, "y"}` in fig. 3.3 aliases the map operator with the label `"y"`.

Given the instruction stream in fig. 3.3, The DAG defined in fig. 3.2 can be recreated by *folding* the instructions from fig. 3.3 on an empty DAG. Consider the pseudo-code in listing 18. The code defines the process the compiler generates to build the modified DAG, given an instruction stream.

Compile-Time Meta-DAG

The instruction stream is the reified representation of a DAG defined at the base-level. The meta-level processes this stream of instructions to modify the

```
DAG = <new DAG>
IS  = <instruction list>

for each instruction I in IS
    case type(I)
        alias ->
            ignore
        operator ->
            DAG.add_vertex(I.type, I.argument)
        edge ->
            FROM = DAG.getVertexByName(I.from)
            TO   = DAG.getVertexByName(I.to)
            DAG.add_edge(FROM, TO, I.from_port, I.to_port)

return DAG
```

Listing 18: *Psuedo-code to fold the instruction stream into a DAG.*

topology of the base-level DAG at compile-time. The user-defined meta-level application that defines the transformation is called the *compile-time meta-DAG*, or *meta-DAG* (step 3 in fig. 3.1).

The meta-DAG is defined using regular Creek. The Creek compiler deploys the meta-DAG at compile-time and emits each instruction (i.e., $i_0,..i_n$) as a meta event through the deployed meta-DAG in the order they appear in the instruction stream. The reflective architecture of μCreek$_C$ reifies two data structures in each emitted meta event. Firstly, each meta event contains an instruction (e.g., d_0). These instructions can be transformed to change the type and arguments of the operator they define. Second, each event contains a reified representation of the DAG that is currently being built (e.g., d_i), called the *working DAG*. The result of propagating an event through the meta-DAG is a working DAG (e.g., d_1). The updated working DAG is paired with the next instruction for the next event. This process is repeated until all instructions have been processed. The final working DAG produced by the compiler is the result.

Figure 3.4 depicts this process. The first instruction from the instruction stream, i_0 is paired with the empty DAG, d_0. This creates the tuple (i_0,d_0), which is "streamed through" the user-defined meta-DAG. The meta-DAG emits a new working DAG, d_1, which is then paired with the next instruction i_1, and the process repeats. When all instructions have been emitted, the result is the final dag d_n.

The only constraint on the topology of the meta-DAG is that it should be a closed DAG with a single source and sink. The meta-DAG should understand

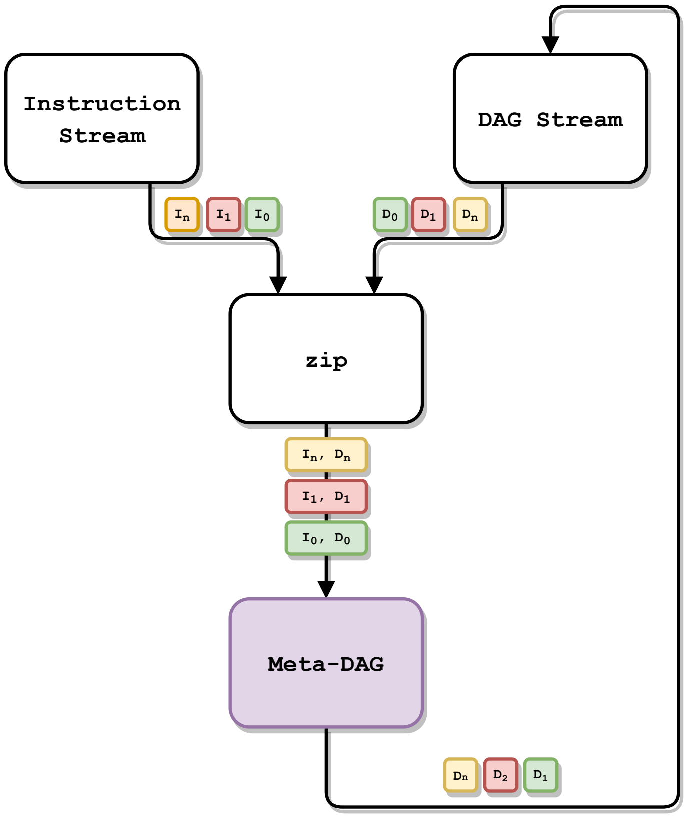


Figure 3.4: *Stream diagram of the evaluation of the meta-stream.*

the full μCreek$_C$ protocol. This entails that the meta-DAG must understand tuples with a DAG and three types of instructions. Additionally, the meta-DAG must also always return a DAG.

μCreek$_C$ Meta Language

The previous two sections introduced the instruction stream and the meta-DAG structure. In this section we introduce the μCreek$_C$ meta language; the language consists of additional language constructs that are available only in μCreek$_R$ DAGs.

The design of a meta-protocol has as its primary goal to be expressive. In μCreek$_C$, a meta-DAG must be able to modify the operators as well as their position in the DAG. Solely transforming instructions does not yield a very expressive meta-DAG. Based on instructions alone, the meta-DAG can only change individual operators and their connections. The working DAG and the alias instructions add another layer of expressivity. Through the working DAG and the aliases, the meta-DAG can fetch previously defined operators and modify those as well. This opens up the possibility make changes to instructions based on previously defined operators, e.g., operator fusion. The following primitives help in modifying previously defined operators.

- `fetch!(name)` fetches the operator instance bound to the name `var` in the working DAG.
- `fuse!(op1, op2)` fuses exactly two existing operator instances together into a new operator instance. For example, if two `map` operators `o = map(f)` and `p = map(g)` exist in the working DAG, `fuse!(o,p)` creates a new operator equivalent to `map(fn v -> g(f(v))}`. Multiple operators can be fused by composing fuse calls to pairs of operators.
- `swap!(op1, op2)` swaps two operators in the working DAG while preserving the connections.
- `inputs(op)` and `output(o[)])` return a list of operators that are directly connected to an input or output port of `op`.

Because the operator instructions alone do not offer enough expressivity, each instruction is paired with the working DAG. The following primitives allow the meta-DAG to modify this DAG using a traditional DAG interface: adding and removing vertices, and placing edges between vertices.

- `add!(op)` and `delete!(op)` add and delete operators, respectively.
- `connect!(op_from, fidx, op_to, tidx)` adds an edge between two existing operators.

- `disconnect!(op_from, fidx, op_to, tidx)` deletes an edge between existing operators.

The primitives described above resemble a regular graph manipulation library. In essence, structural reflection of streams boils down to just that. However, under the hood, connections between ports and identities need to be managed, and structural invariants (e.g., well-formedness) must be preserved. The primitives described above disallow manipulating the internal data structures. The only way to manage the working DAG is by means of the predefined functions. The abstractions defined above are designed with expressivity in mind, while hiding the details of the DAG construction.

The user-defined meta-DAG processes tuples consisting of an instruction and a working DAG, and must emit a working DAG. This entails evaluating the instruction and making the necessary changes to the working DAG, using the functions defined above. To avoid requiring the simplest of meta-level applications to evaluate instructions, μCREEK$_C$ offers a built-in open DAG, `proceed`. The `proceed` DAG takes as input a tuple of instructions, and a working DAG, and emits an updated working DAG. In a sense, the `proceed` DAG is the default meta-level behavior. Figure 3.5 shows the diagram of the `proceed` operator. For each tuple that comes in, a modified working DAG is emitted.

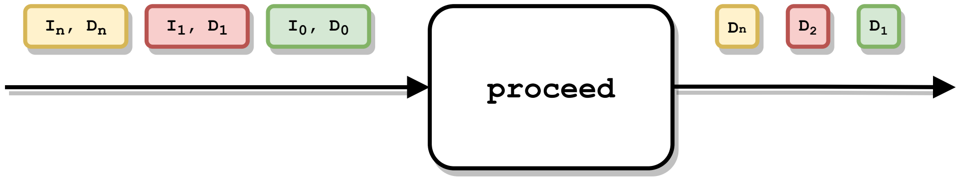


Figure 3.5: *Diagram for the built-in* `proceed` *DAG.*

Compile-Time Guarantees μCREEK$_C$ allows the programmer to transform the compilation instructions at compile-time. Because the language does not prohibit the programmer from violating the constraints mentioned in section 2.2.3, an additional step is transparently executed at compile-time to ensure that the μCREEK$_C$ program did not generate an invalid DAG. If the μCREEK$_C$ program produces an invalid DAG the program will be rejected by the compiler.

Example Listing 19 shows a complete example of a meta-DAG, `VerboseMap`. The program modifies every `map` operator in the base-level DAG by replacing its argument function with a wrapped function that prints out its input and output to the console. The meta-DAG duplicates its input data over two open DAGs: `operator` and `others`, defined in lines 4–20. The `operator` DAG filters out all instructions that are not operator instructions and transforms them using a `map` operator. If the operator is of type `:map`, its argument is wrapped into a new anonymous function that prints a message to the console (lines 7–15). Any other operator in the DAG is left as is. The `others` DAG filters out all operator instructions. Streaming any instruction stream through `metadag` results in a DAG that is structurally different in the sense that each operator in the DAG that is of the type `:map` has its argument replaced with a wrapped function. In section 3.4.1 we show more complex examples of meta-DAGs such as operator fusion.

```
 1 defmodule VerboseMap do
 2   use Creek
 3 
 4   defdag operator as filter(&(match?({{:operator, _, _}, _, _}, &1)))
 5                      ~> map(fn {{:operator, type, arg}, dag, it} ->
 6                          arg =
 7                            case type do
 8                              :map ->
 9                                fn x ->
10                                  IO.puts "f(#{x})"
11                                  res = arg.(x)
12                                  IO.puts "=> #{res}"
13                                  res
14                                end
15                              _ ->
16                                arg
17                            end
18 
19                          {{:operator, type, arg}, dag, it}
20                        end)
21 
22   defdag others as filter(&(not match?({{:operator, _, _}, _, _}, &1)))
23 
24   defdag metadag(src, snk) do
25     src
26     ~> dup
27     ~> (operator ||| others)
28     ~> merge
29     ~> proceed
30     ~> snk
31   end
32 end
```

Listing 19: *Meta-program that logs the input and output of each map operator in a base-level DAG.*

A base-level application can declare a meta-DAG with the **`structure`** pragma. The structure pragma takes the name of a module as a parameter. Every DAG defined in the module is compiled with the meta-DAG defined in the given module. An example of a program that uses the `VerboseMap` is shown in listing 20.

```
1  defmodule ExtractJson do
2    use Creek
3    structure VerboseMap
4
5    defdag extract_ip(src, snk) do
6      src
7      ~> map(fn bin -> JSON.parse(bin) end)
8      ~> map(fn dict -> Map.value(dict, :ip_address) end)
9      ~> snk
10   end
11
12   # ...
13 end
```

Listing 20: *A DAG that extracts IP addresses from binary data. The DAG is compiled with verbose logging (line 3).*

Multiple μCreek$_C$ DAGs can be composed. For example, replacing `structure VerboseMap` with `structure VerboseMap, OperatorFusion` in listing 20 first adds logging statements, and then fuses operators together. Conceptually speaking, a μCreek$_C$ programs takes in a DAG, and produces a DAG. The resulting DAG can again be processed by another μCreek$_C$ DAG. When multiple μCreek$_C$ DAGs are defined, the meta DAGs are applied from left to right.

In conclusion, a meta-DAG can modify the structure and operators of a base-level DAG at compile-time. The goal of the DSL is to build DAG transformations in the form of DAGs that can be composed into specific compile-time behaviors. This is achieved by reifying the base-level DAG as an instruction stream and a DAG that can be manipulated using the μCreek$_C$ language.

Validation

To validate our approach, we implement operator fusion as an example of a compile-time transformation. Operator fusion optimizes a DAG by replacing multiple operators with a single one. Semantically the result is equivalent, but the amount of operators, thus increasing performance and reducing the DAG's size. We consider operator fusion a validation because it uses all the features present in μCreek$_C$: 1) removing and swapping operators in the DAG at compile-time, 2) inserting operators at compile-time, and 3) changing the connections between operators in the DAG based on stream events, and 4) is was identified as a shortcoming by related work [17, 30]. Operator fusion subsumes problems such as logging, timestamping, and parallelizing a stream. At the end of this section, we briefly discuss the approach to implement these compile-time transformations in *μ*Creek$_C$. Their implementation can be found in appendix B, and appendix D.

Listing 21 shows the relevant code required to implement the fusion of consecutive `map` operators in *μ*Creek$_C$. Operator fusion is implemented by transforming the DAG based on the edge instructions. The alias and operator instructions are not transformed and therefore the code is not shown here. The full code can be found in appendix C.

Fusing two operators together at compile-time in *μ*Creek$_C$ is possible because each operator is reified as an instruction that contains the operator type (e.g., `map`, `filter`,...), and as its arguments (i.e., the function of a `map` operator). Fusing two `map` operators involves wrapping the arguments of the individual operators into a single function, which is the argument of a new `map` operator.

Given an edge between two `map` operators *a* and *b*, it is possible to fuse *a* and *b* into *c*, and connect the input of *a* to *c*. All the subsequent edge instructions that define an edge going from *b* to another operator must be replaced by an edge going from *c* to the other operator.

The `edge` DAG defined in listing 21 filters out all instructions, except the edge instructions. First, the operators of the edge are fetched. If those two operators are not `map` operators, the instruction is left unmodified (lines 17–18); otherwise it is modified as follows.

1. A new operator *c* is created with a function that applies *b* after *a* (line 12).
2. Operators *a* and *b* are deleted from the working DAG (lines 13 and 14).
3. Operator *c* is inserted into the working DAG (line 15).
4. The original instruction is transformed to create an edge between the output port of the source of *a* and the input port of *c* (line 16).

All other events (i.e., operators and name events) are propagated through the meta-DAG without modification. Combining the `proceed` DAGs with the `edge` DAG results in a closed DAG that can be plugged into the compiler to achieve the desired behavior. We refer to appendix C for the full implementation.

Timestamping Timestamping the values of a stream means that each datum is timestamped at the time it passes through an operator in a DAG. This can be achieved in contemporary stream DSLs by adding a `map` operator that adds a timestamp to each value. However, inserting `map` operators that wrap values

```
defdag edge as
        filter(fn event ->
            match?({{:edge, _, _, _, _}, _, _}, event)
        end)
        ~> map(fn {{:edge, from, fidx, to, toidx}, dag, it} ->
            a = fetch!(dag, from)
            b = fetch!(dag, to)

            case {a.name, b.name} do
                {"map", "map"} ->
                    [x] = inputs(dag, a)
                    c = fuse(a, b)
                    dag = delete(dag, a)
                    dag = delete(dag, b)
                    dag = add!(dag, c)
                    {{:edge, x.ref, 0, c.ref, 0}, dag, it}

                _ ->
                    {{:edge, from, fidx, to, toidx}, dag, it}
            end
        end)
```

Listing 21: *Example of a meta stream fusing consecutive map operators together.*

with timestamps impacts every other domain logic operator. The original operators must now be adapted to work with timestamped values, introducing additional boxing and unboxing logic.

Using μCreek$_C$, timestamping can be implemented by making two changes to the DAG at compile-time. First of all, a timestamp operator is placed placed in front of every base-level operator. Second, every operator that has a function argument its argument is wrapped by a function that boxes and unboxes each datum. The result is that a stream is timestamped, but the operators to do so are extracted from the domain logic. Figure 3.6 shows two diagrams that represent this transformation.

Parallelization Parallelizing a DAG or operator means that single operators are executed in parallel. Instead of processing one value at a time, they can now process multiple values. Figure 3.8 and fig. 3.7 show how this is achieved in Akka Streams by duplicating operators in the DAG.

Similar to the approach for operator fusion, a `map` operator can be replaced with a set of other operators: a `balance` operator that spreads its data over n instances of the original `map` operator, and a `merge` operator which merges those parallel streams. This approach leads to the same topology as in fig. 3.8, but is generated by the compiler, instead of by hand. Just like the Akka Stream and

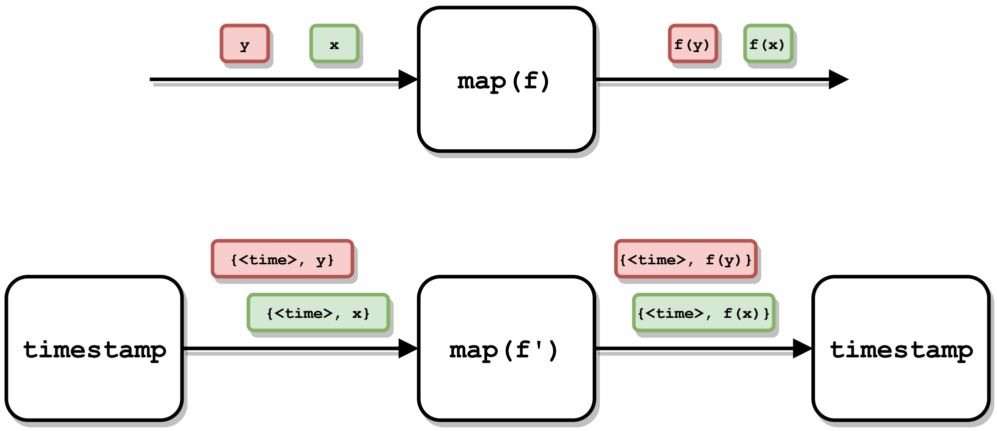


Figure 3.6: *Diagrams of a regular map operator, and a timestamped map operator.*

```
source
.map(compute)
.deploy()
```

Figure 3.7: *Akka Stream DAG to apply compute on each datum.*

```
val processor =
  Flow.fromGraph(GraphDSL.create() { implicit b =>
    val balance = b.add(Balance[Int](2))
    val merge = b.add(Merge[Int](2))
    val f = Flow[Int].map(compute)

    balance.out(0) ~> f.async ~> merge.in(0)
    balance.out(1) ~> f.async ~> merge.in(1)
    FlowShape(balance.in, merge.out)
})
source.via(processor).runForeach(println)
```

Figure 3.8: *The application from fig. 3.2 parallelized wih a factor of 2.*

RxJava implementation in listing 14, our approach loses the original ordering of the data due to race conditions.

3.4.2 μCreek$_R$: Run-Time Meta

The compile-time approach proposed in section 3.4.1 makes it possible to express structural changes to the DAG. This helps to express non-functional concerns such as logging, parallelism, etc. However, some of the problems introduced in section 3.1 can not be addressed with topology changes alone using μCreek$_C$. Some problems require intercession at run-time. Examples of these problems are the propagation semantics of a stream, encryption, and load balancing.

We introduce – complementary to μCreek$_C$ – a *behavioral reflection* architecture. *Behavioral reflection* allows a program to modify the run-time semantics of the underlying language itself. In the context of stream DSLs this means that streams can manipulate how they are executed at run-time. For example, behavioral reflection can introduce different propagation semantics, encrypt an entire stream, or catch errors to avoid retrying a stream by intercepting the execution of event-handlers.

In the remainder of this section we introduce the architecture of *behavioral reflection* for a stream DSL by adding it to our own stream DSL Creek (section 2.2) and discuss the intercession language of μCreek$_R$ (section 3.4.2).

μCreek$_R$ Architecture

When a stream is deployed, its individual operators are exchanging internal messages under the hood (section 2.2.5) to propagate the data. This protocol is called the Canonical Stream Protocol (CSP). *Behavioral reflection* gives the programmer control over this process by means of a reflective architecture. The messages from the CSP are called *base messages*, e.g., `{:next, 7}` which means that in a stream of integers, 7 was emitted by an upstream operator. In the case of a stream of integers, 7 is a *domain value*.

User-defined DAGs do not deal with base-level messages directly; base-level messages are internal communication of the runtime. The runtime handles base-level messages by calling the appropriate event-handler in each operator, and depending on its result, sending the appropriate messages to the up- and downstream operators, i.e., the CSP. For example, when a `map` operator receives the base-level event `{:next, 5}`, the runtime executes the following steps. First, the `next` event-handler is called from the operator. Second, depending on the result of the event-handler, the value is propagated to all the downstreams, or an error or complete event are sent downstream. Figure 3.9 graphically depicts how base-level messages are processed at the meta-level.

The meta-level applications, called *meta-DAGs*, are user-defined DAGs that consume meta messages. Because meta-DAGs hook into the runtime, they must fulfill the tasks of the runtime. For each meta event the meta-DAG must call the appropriate event handler of the operator, and ensure that the right internal messages are sent to the up- and downstream operators. Calling the correct event handler is called the *base stage*, and sending the appropriate internal messages is called the *effects stage*. In the meta-DAG there is no clear distinction between these two stages. The meta-DAG simply has control over these two mechanisms. However, the effects stage includes message sending. Once these side effects have been done, they cannot be undone.

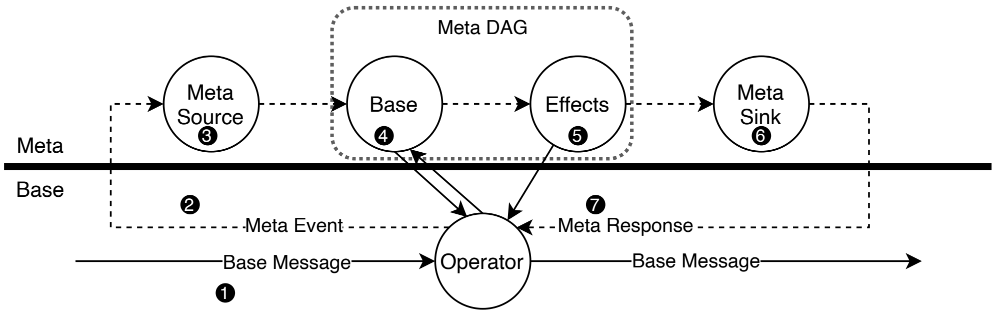


Figure 3.9: *Architecture of a meta-level stream: each base message is reified into the meta-level, and each meta response is deified to the base-level.*

Figure 3.9 depicts a flow diagram of how a base event is processed. The process starts when a base event arrives at the base-level ❶. The runtime reifies the base message to a meta-level representation ❷ – a meta event – and emits it through its meta source ❸. The meta-DAG has two distinct stages: base-level and effects. First, the stream calls the relevant base-level function for handling the meta-level event ❹. Second, the stream emits the necessary side effects in response to the reply of the event-handler ❺. For example, in response to `{next, v}`, a value must be propagated to all operators connected to the output ports. The meta response of the stream, representing the new operator state, is captured by the sink actor ❻. And finally, the result of the meta stream is installed as the new operator state ❼, and the operator can receive a new base-level message.

μCreek$_R$ reifies base-level messages into the meta-level as *meta messages*. These messages are emitted by a *meta source*. The internal messages have been discussed at length in section 2.2.5. We will not discuss each specific meta event here because they are highly similar to the CSP. Instead, we explain the reification process, and highlight the differences. An internal message contains the *event type* (e.g., `error` and `next`) and the input port on which it arrived. The reification of a message adds the following data.

1. **Operator State** The operator state contains the state of the operator that is available in the behavior of the operator. For example, the `from_list` source operator keeps the values it must emit in its state.
2. **Meta State** The meta-state can be used to store data across multiple meta messages. For example, the pull-semantics implementation makes each operator keep track of which operators it has asked data of, and which operators have sent data in response to avoid sending demand twice.
3. **Operator** The reification of the operator is a representation of the operator containing its type (e.g., `map`), arguments, and unique label.
4. **Up- And Downstream** A list of reified representations of all up- and downstream operators. These lists can be used to selectively send messages downstream or upstream. For example, the `balance` operator only forwards a message to a single downstream operator. Most operators, e.g., the `dup` operator, forward values to all downstream operators.

Every meta event must be turned into a *meta response* by the meta-DAG. A meta response contains the updated operator state and meta state, and the up- and downstream operators. All the other information is irrelevant to process the next meta event. μCreek$_R$ provides a *meta sink* for the meta responses. Each meta response is handled by installing the new operator and meta state and updating the down- and upstream operators.

μCreek$_R$ Language

The previous section introduced the architecture of μCreek$_R$. The architecture exposes the internal messages of the stream to the meta-level as meta events. These meta events can be processed by a user-defined meta-DAG to change the default semantics of the stream. A meta-DAG must handle two concerns. For each meta event, the meta-DAG must call the base-level event-handlers (i.e., base stage), and send internal messages to connected operators (i.e., effects stage).

Once an internal message has been reified to the meta level, both stages are be handled at the meta level. Depending on the non-functional requirements, not every meta-DAG modifies the default semantics in both stages. To avoid rewriting the default semantics in the meta-DAG, two built-in DAGs are provided by μCreek$_R$: `proceed` and `effects`. The `proceed` DAG defines the default semantics for the base stage, and the `effects` DAG defines the default seman-

Function	Description
propagate_up(e)	Emit e to all the operators connected to an input port.
propagate_down(e)	Emit e to all the operators connected to an output port.
propagate_self(e)	Emit event e to itself.
call_base(name, arg*)	Calls the event-handler name with optional argument arg. name can be next, error, complete,tick or initialize
state(op)	Returns the current state of the operator.
set_state(op)	Updates the state of the operator, and returns a new operator.
meta_state(op)	Returns the meta state of the operator.
set_meta_state(op)	Updates the meta state of the operator, and returns a new operator.

Table 3.1: *Functions available in μCreek$_R$*

tics for the effects stage. This allows a meta-DAG to only define changes in one of both stages, while keeping the default semantics for the other stage. Using these built-in meta-DAGs is optional because a meta-DAG can redefine both stages. The default meta-level semantics are captured in the DAG `proceed ~> effects`.

The user-defined meta-DAGs can be constructed with all the base-level operators found in Creek. The only constraints put on meta-DAGs is that they understand all meta messages and emit meta responses, and that the DAG is a closed DAG with an actor socket for 1 source and 1 sink. These actors sockets are replaced by the meta source and meta sink at runtime.

A user-defined meta-DAG can manipulate the way the base-level is called, and what side effects happen. μCreek$_R$ offers abstractions to manipulate this behavior. While the abstractions expose a simple message send at the meta level, the internals require some additional bookkeeping (e.g., adding process identifiers and output ports). Table 3.1 summarizes the functions available to the operators of a meta-DAG. Three functions are available to handle side effects (i.e., `propagate_*`), a getter function for the operator state (i.e., `state`), and a function that calls the base level event-handlers (i.e., `call_base`).

Example As an example of how all these concepts are tied together, we show a logging meta-DAG as an example of a μCreek$_R$ application (listing 22), a base-level application that loads it (listing 23), and a sample output of running the application (listing 24).

The code shown in listing 22 logs all the values that are received by and emitted by an operator. An open DAG `log_next` is created to handle all the `next` meta events. For each next event a message is logged to the console, printing out the value that arrived (lines 13–16). Next, the meta event is handled by the `base` DAG, because the meta application does not intend to modify the default semantics. The `base` DAG returns the instruction from the operator behavior.

Depending on that result the appropriate message is logged to the console (lines 18–31). Finally, the base response is emitted to the `effects` DAG to send the appropriate messages.

The application shown in listing 23 shows how to calculate the stream of factorial numbers. The application can enable logging for the defined DAGs by using the `behavior Logging` statement on line 3. An example of the output of the logging DAG is shown in listing 24.

Validation

To validate μCreek$_R$, a form of pull-based propagation semantics is implemented in Creek, which is push-based by default (see section 2.2). We consider an alternate form of propagation semantics a validation because 1) related work [17, 30] posits it as one of the key shortcomings of contemporary stream DSLs, and 2) it requires changes in the default semantics across the entire stream (i.e., sources, sinks, and operators).

Creek implements push-based semantics by default. Any source operator emits a new value in response to a `tick` event. These tick events are sent on two occasions. The first `tick` is sent when the stream is deployed, to start the stream. Subsequent `tick` events are sent whenever a source has emitted a value, to signal producing the next. This loop of `tick` events constitutes *push-semantics*, because the source does not take the downstream into account. Figure 3.10 graphically depicts the process. The `A` operator is a source that continuously receives `tick` events.

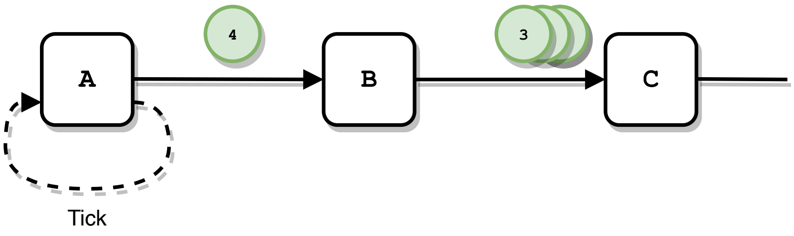


Figure 3.10: *Diagram depicting the "tick loop" in push-based semantics.*

A stream with *pull-semantics* does not allow a source to dictate the rate of emission. The source operators no longer sends `tick` messages to themselves, and the `tick` message is no longer sent when the stream is deployed. The essence of pull-based semantics is the addition of the `demand` meta message. The sinks of the stream send the initial `demand` message upstream when the stream is deployed, signaling they require data. In what follows, we describe

```
defmodule Logging do
  use Creek
  use Meta

  defdag default as
    base ~> effects

  defdag passthrough as
    filter(&(not match?({_, :next, _, _}, &1)))
    ~> default

  defdag log_next as filter(&match?({_, :next, _, _}, &1))
                      ~> map(fn {op, :next, v, frm} ->
                        IO.puts("#{op.name} incoming: #{v} from #{frm}")
                        {op, :next, v, f}
                      end)
                      ~> base
                      ~> map(fn result ->
                        case result do
                          {op, state, :next, result} ->
                            IO.puts("#{op.name} outgoing: #{result}")

                          {op, state, :error, err} ->
                            IO.puts("#{op.name} error: #{err}")

                          {op, state, :complete} ->
                            IO.puts("#{op.name} completed")
                        end

                        result
                      end)
                      ~> effects

  defdag meta(src, snk) do
    src
    ~> dup()
    ~> (log_next ||| passthrough)
    ~> merge()
    ~> snk
  end
end
```

Listing 22: *Meta-DAG to log all the incoming values and outgoing values of each operator in a stream.*

```
defmodule LoggingExample do
  use Creek
  behavior Logging

  defdag factorial(src, snk) do
    src
    ~> transform(1, fn v, acc -> {acc * v, acc * v} end)
    ~> snk
  end

  def main() do
    source = Source.range(1, 10)
    sink = Sink.all(self())
    deploy(factorial, [src: source, snk: sink])

    receive do
      result ->
        # Handle result..
    end
  end
```

Listing 23: *Example application that loads the* `Logging` *meta-DAG.*

```
...
Source    outgoing: 10
transform incoming: 4 from #<InputGate>
transform outgoing: 24
transform incoming: 5 from #<InputGate>
Source signaled to produce!
transform outgoing: 120
Source    outgoing: 11
Source signaled to produce!
...
```

Listing 24: *Example output of the* `Logging` *meta-DAG.*

the meta-level behavior for sources, sinks, and operators.

Source Listing 26 shows part of the implementation of pull-based propagation semantics in Creek, namely the meta-DAG for a source operator. The full implementation can be found in appendix E. The changes required for a source operator are three-fold. First, when the `init` message is sent when the stream is deployed, no effects are executed. These effects send the initial `tick` message, and this is not needed in pull-semantics. Listing 25 shows the open DAG that handles all the `init` messages for a source. The `base` DAG is used to call the appropriate behavior, but the effects are removed to avoid sending the `tick` message.

```
defdag init_src as
    filter(&match?({_, :init}, &1))
    ~> base()
    ~> map(fn {p, state, :initialized} ->
        # Here we would normally send tick to ourselves, but we dont.
        {p, :ok}
    end)
```

Listing 25: *Handling `init` messages for source operators in pull-based streams.*

Second, when a source receives a `tick` event, it produces a value and sends a `tick` message to itself. Again, this is done by overriding the effects of the `tick` message. Listing 26 shows the open DAG that handles all `tick` messages sent to a source operator. The `base` DAG is used to call the appropriate behavior, but the effects are modified. Depending on the result of the base level, the necessary effects are executed. When the base level returns a `tick` instruction, the source propagates its data downstream, but does not send the `tick` message.

Finally, the source operator must understand the `demand` message. The operator sends itself a `tick` message in response to each `demand` message. The `demand` message is not part of the CSP, and therefore there is no event handler defined in the behaviors.

Operator If an operator receives the `demand` message, it has to propagate it upstream such that it eventually reaches a source. Some operators, such as the `filter` operator, do not always emit a value in response to incoming value. For example, a filter for even numbers will not emit a value if it received an odd number in response to the `demand` message. To solve this problem, every operator must keep track of the `demand` messages it sends. If the operator does not emit a value downstream in response to an upstream value, it must send `demand` again, to satisfy the demand of its downstream. This is further

```
defdag tick_src as
    filter(&match?({p, :tick}, &1))
    ~> base()
    ~> map(fn base_result ->
        case base_result do
        {p, {state, :complete}} ->
            effects_complete(nil, p.ds, p.us, p.pid)
            {%{p | state: state}, :ok}

        {p, {state, :next, value}} ->
            # Here we would normally send tick to ourselves, but we dont.
            propagate_downstream({:next, value}, p.ds, p.pid)
            {%{p | state: state}, :ok}
        end
    end)
```

Listing 26: *Handling `tick` messages for source operators in pull-based streams.*

```
defdag demand_src as
    filter(&match?({p, :demand, _}, &1))
    ~> map(fn {p, :meta, :demand, _} ->
        send_self({:tick}, p.pid)
        {p, :ok}
    end)
```

Listing 27: *Handling demand messages for source operators in pull-based streams.*

complicated in case of operators with multiple upstreams. Operators that rely on multiple upstreams (e.g., `zip`) maintain state to keep track of which sources have been sent demand and which sources replied. This avoids sending demand twice to an operator.

Listing 28 shows the open DAG that defines how a demand message is handled by an operator. The meta state contains a dictionary that contains which upstreams received a demand message already. If the operator receives a demand message, it is forwarded to all upstream operators that have not been sent a demand message only.

Sink When a sink is deployed with the default semantics, it does not execute any effects in response to the `init` message. In case of pull-semantics the sinks are the operators that have to send the initial demand message. Additionally, when a sink receives a value from its upstream is should send a new demand message upstream The full implementation for pull-based semantics can be found in appendix E.

```
defdag forward_demand as
    filter(&match?({_, :demand, from}, &1))
    ~> map(fn {opr :demand, from} ->
        demanded = opr.meta_state
        to_demand =
            opr.upstreams
            |> Enum.filter(&(not MapSet.member?(demanded, &1)))
        propagate_upstream(:demand, to_demand, opr.pid)
        meta_state = MapSet.new(opr.upstreams)
        {%{p | meta_state: meta_state}, :ok}
    end)
```

Listing 28: *Handling demand messages for source operators in pull-based streams.*

3.5 Discussion

The design of a meta-level architecture should respect certain design principles set forth by earlier work in meta-level programming. One of the most prominent work in this field is the work by Bracha and Ungar [20] and Kiczales et al. [59]. We discuss the principles put forward in this work, and discuss why μCreek was designed according to these principles, and if μCreek upholds them.

Encapsulation dictates that a meta-level architecture should *encapsulate* its implementation details. The meta-level programs should be written against an API that decouples the underlying implementation from the meta-level programs, ensuring reusability. For μCreek$_C$ we have chosen to write meta-programs against the instruction stream that exposes three data structures (Directed Acyclic Graphs (DAGs), edges, and operators), offering an API that is completely decoupled from the actual implementation. The result is that μCreek$_C$ programs can be reused for implementations in other languages, as long as the meta-level architecture offers the same API.

For μCreek$_R$, a meta-DAG can intercept at the level of the communication protocol defined in section 2.2.5. Each message is reified in such a way that it hides details regarding communication protocols, concurrency, etc. From the meta-level perspective, an operator is a black box that offers a few behaviors, and that has state. This approach almost completely decouples the μCreek$_R$ program from the base-level, as no references can be made to the base-level details, but only reified representations thereof. The only requirement for any language to support μCreek$_R$ is to reify the inter-operator communication as the canonical stream protocol and to reify the representation of base-level operators their behaviors.

Stratification dictates that meta-level entities should be cleanly separated from base-level entities. This ensures that there is as little coupling as possible between the base- and meta-level and that the meta-level behavior can be removed without breaking the base-level program. A meta-level program should not reference values or entities from the base-level (e.g., a base-level DAG with variable name `proceed` can not have impact on the `proceed` DAG at the meta-level), or vice versa. Additionally, a base-level operator should be prohibited from creating explicit references to its meta-level representation. These references would make it impossible for the compiler to completely remove the meta-behavior because the base level is coupled to the meta-level. If stratification is respected, the meta-level architecture can be completely removed from an application if it is never used, without changing the semantics of the application.

In μCreek$_C$ stratification is ensured in three ways. First, the meta architecture is only loaded on-demand by using the `structure` pragma. If no μCreek$_C$ program is installed in the base level program the compiler will not activate the meta-DAG compiler. Second, the meta-level does not have access to base-level concepts but only the instruction stream, and thus no references can be made to base-level entities. Third, the Domain Specific Language (DSL) for the meta-DAGs is namespaced, ensuring that the `proceed` and `base` DAGs cannot be confused with base-level DAGs.

Similarly, in μCreek$_R$, the meta-program is only loaded if it has been explicitly imported using the "`behavior`" pragma. If there is no pragma defined the compiler will not install any meta-level behavior in the operators to modify the runtime behavior. Additionally, the μCreek$_R$ DAG does not have access to base-level concepts, except values from the host language.

Structural Correspondence dictates that each language construct has a reified representation at the meta-level. μCreek$_C$ represents the compile-time components of a stream language at the meta-level: DAGs, edges, and operators. While strictly speaking the function arguments of operators should also be reified, we have chosen not to do so, as this requires a meta-level representation of concepts from the host language (i.e., Elixir), coupling μCreek$_C$ to a specific implementation. We have reified only the concepts that μCreek introduced and the host-level language. Similar to μCreek$_C$, we have chosen to only represent the run-time concepts introduced by the μCreek DSL at the meta-level, i.e., operators and events in the Canonical Stream Protocol (CSP). A μCreek$_R$ application is a transformation of incoming and outgoing meta events. The meta-level applications do not have a notion of the stream as a whole, only the operator on which it is deployed. Therefore streams are not reified at the meta-level.

Unified Programming Model A unified programming model entails that the paradigm for the meta-level and the base-level are the same (e.g., streams). Matching these paradigms has a number of advantages. On the one hand, it simplifies reasoning over the code as the same programming concepts are used in both meta and base level code. On the other hand, reification simplifies the interaction between both levels, because the stream-paradigm does not have to be translated to another paradigm.

For this reason, μCreek$_C$ represents the construction of the DAG as a stream of instructions. This also means that μCreek$_C$ programs are streams that are built using the full expressivity of the Creek language. μCreek$_R$ represents the base-level events as meta-events that are streamed from the meta-source. The μCreek$_R$ programs manipulate these meta-events, and in doing so, have full control of the semantics of the semantics of the stream. **We conclude that the paradigm of the base-level (i.e., Creek) , compile-time meta-programming (i.e., μCreek$_C$), and run-time meta-programming (i.e., μCreek$_R$) are the same.**

Portability To enable the implementation of the ideas of μCreek in other languages, the meta-level architecture cannot rely on the intricacies of the host language. The design principles outlined by Bracha and Ungar (i.e., encapsulation, stratification, and structural correspondence) set guidelines to prevent a meta-level architecture from becoming too entwined with its base-level. However, Creek is built atop a concurrent, multithreaded runtime, namely Elixir. If the runtime relies too much on these properties it can become tightly coupled to the execution semantics of the host language (e.g., relying on concurrency). In μCreek$_R$, the meta-level programs only process events emitted by the runtime (i.e., reified base-level events). While not every language implements the communication protocol of Creek, we argue that the message protocol presented in section 2.2.5 is sufficiently high-level to express the internal messages of most languages. To port the architecture of μCreek to another language, the runtime must generate and emit the events from the μCreek protocol and understand the messages it can generate (table 3.1). Additionally, the protocol defined in Reactive Extensions[82] was used as inspiration to the μCreek protocol and is the foundation of most JVM-based stream languages [83, 66] and other reactive extensions implementations, meaning that the internal protocol is similar to Creek's.

3.5.1 Design Philosophy

μCreek is designed with the principles of [20] at its core. These design principles ensure that the meta-level architecture is portable, modular, and extensible.

Encapsulation ensures that a meta-level application does not depend on the internals of the base language it is about. In context of μCreek, μCreek$_R$ its domain is meta-events and meta-responses, and is not about Elixir actor messages, processes, or modules. If encapsulation should be violated, the design of μCreek would not be applicable to non-actor languages or languages other than Elixir, without major modifications. *Encapsulation ensures that the meta-level application is portable across implementations.*

Stratification ensures that the meta-level architecture does not impose any performance overhead when not used because the runtime can completely separate application and meta code. In context of μCreek, μCreek$_C$ does not impose any overhead when it is not used by completely disabling it at the virtual machine level. The instruction stream is only generated when a μCreek$_C$ directive is present in the program. Additionally, μCreek$_R$ does not impose any overhead when no directive is present in the program, because a Creek program is completely unaware of its meta-level stream, and vice versa. This stratification makes it possible for the virtual machine to completely disable the meta-stream. The runtime does not load any meta-level streams and does not generate meta-events if no meta-DAG is present. *Stratification ensures that the absence of meta-behavior does not negatively impact the performance of Creek.*

Structural Correspondence ensures that the base-level concepts of a program have a meta-level representation. In context of μCreek$_C$, the meta-application reasons about the structure of DAGs, and the dependencies between the operators in those DAGs. The concepts of a DAG, operator, and dependency are reified at the meta-level. However, values and concepts from the language in which μCreek$_C$ is embedded (i.e., Elixir) are not reified. True structural correspondence would reify these values as well, but the host language (i.e., Elixir) does not support this, thus imposing a significant engineering effort. In μCreek$_R$ the concepts that are added by Creek (i.e., operators, streams, and events) are reified, but similar to μCreek$_C$, values from the host language are not reified. *Structural correspondence ensures that the meta-level can reason about the same concepts as the base-level.*

Unified Programming Model requires that the paradigm at the base-level and meta-level are the same. A unified programming paradigm offers the following advantages.

First, when the base and meta paradigms match, the programmer can reason about base-level and meta-level in terms of similar concepts. Because of this property, the meta-level in both μCreek$_C$ and μCreek$_R$ are stream-based. Second, in the context of embedded DSLs such as Creek, the host language has its own paradigm (e.g., functional), next to the paradigm of the embedded DSL (e.g., stream). A meta-level architecture does not have to employ the paradigm of its host language, or the DSL. A unified paradigm avoids the need for a third paradigm, which would introduce additional cognitive overhead. Third, when meta-programs are written in the same language as the base-level, *reflective towers* become possible. Reflective towers allow the meta-level programs to have a meta-level, which can in turn have their own meta-level, and so on. In μCreek, reflective towers can be used to modify the behavior of μCreek$_R$ and μCreek$_C$ programs. As example, consider the implementation of operator fusion (appendix C). Instead of optimizing the DAG to have as few operators as possible, operator fusion can be used to fuse operators to increase performance.

When the base- and meta-level paradigm do not match, *language symbiosis* between the base- and meta-level is necessary [97]. Symbiosis introduces additional complexity into the meta-level design that has to be addressed [51], depending on the combination of languages. Data structures from the base-level and meta-level must have a representation in both paradigms, and must be translated between them. To explain this, assume that the meta-level of μCreek$_R$ is expressed in the object-oriented paradigm. Every operator emits a continuous stream of meta-events, and expressing logic that reacts to these events requires solutions such as the Observer pattern. Stream-based programming is designed to express event-driven systems, and μCreek$_R$ is an event-driven architecture, so it is evident that stream-based programming is the paradigm at the meta-level. The story is different in context of μCreek$_C$, however. A μCreek$_C$ program is a DAG transformation at its core, and could more easily expressed in an imperative or functional paradigm. However, the advantage of expressing μCreek$_C$ programs in the stream paradigm outweigh the advantages of adding another paradigm. *The unified programming paradigm keeps the cognitive overhead to a minimum and avoids the need for language symbiosis between the meta-level and base-level.*

3.6 Evaluation

We addressed how μCreek can address the problems postulated in section 3.2. In this section we quantitatively measure the performance overhead of μCreek$_R$ by benchmarking μCreek against a version without meta-level architecture. We also showcase *Anabranch*, a debugger for streams, written completely at the meta-level, with the purpose of showing the expressiveness and limitations of the meta-level architecture.

3.6.1 Creek Debugger

Meta-level architectures open the language in a structured way. The reification of the internal messages in the language exposes information typically inaccessible to the programmer. A *debugger* is a well-known tool that can use this information to give additional insights into the execution of a program, or manipulate it. In this section, we discuss *Anabranch*, a reflective debugger for Creek in the form of a meta-Directed Acyclic Graph (DAG). Anabranch is used to show the expressiveness of our meta-level architecture.

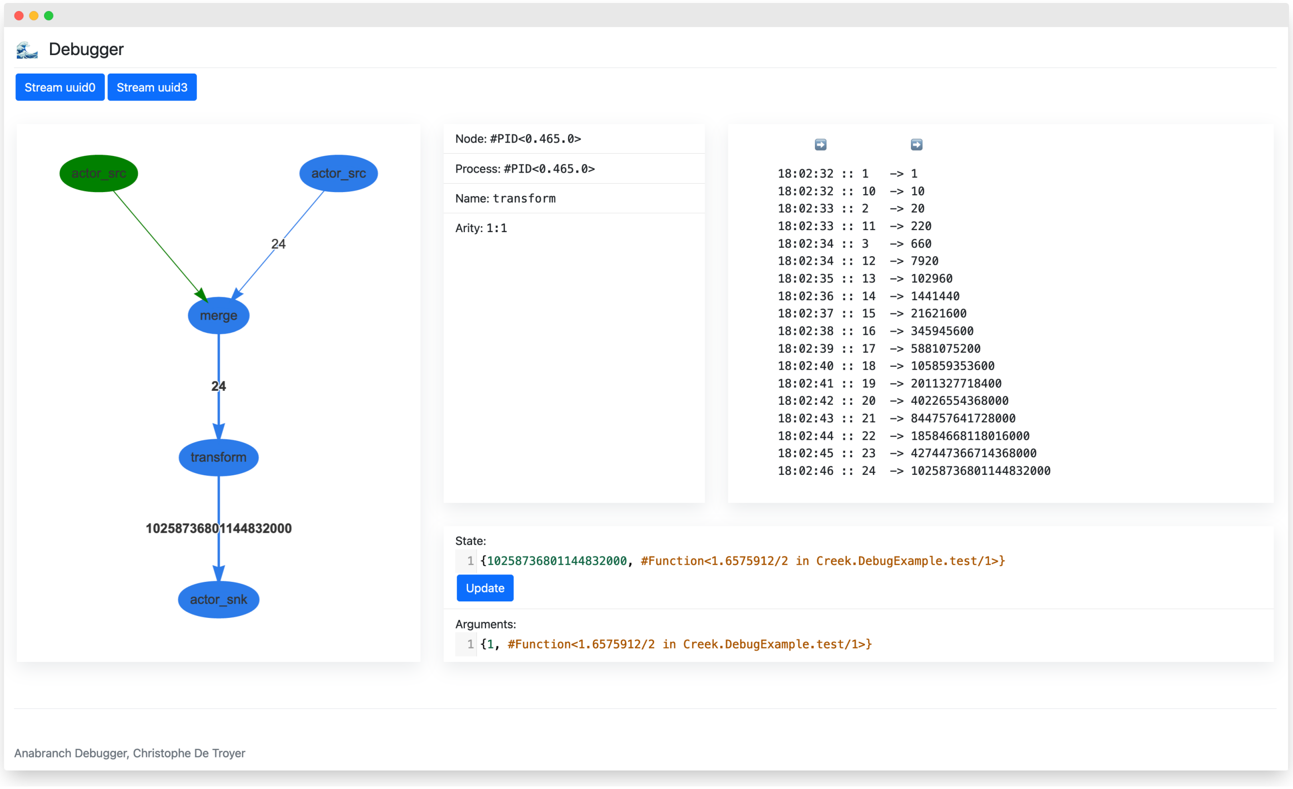
A debugger is a program that allows the controlled execution of a *target program*. A minimal requirement for debuggers is to give additional insight into the target program's execution. Typically, a debugger allows tracking a variable's value, modifying the target program while running, pausing the execution of a program, stepping the execution of the program, and evaluating expressions in the current environment. There are two major types of debuggers: *online* and *offline* debuggers, also called *post-mortem debuggers*. Online debuggers execute while the program is running, and offline debuggers work based on logs, traces, or dumps of the program or checkpoints in the program. *Online debuggers* are debuggers that execute while the program is executing. Typically these debuggers are used by placing breakpoints in the code, where the debugger should halt the program for further inspection. A debugger consists of two parts. The *front-end* is the interface that programmers use to interact with the program. The *back-end* is the software that controls and manages the program execution. In online debugging, the *front-end* can run remotely, or in the same process as the application, called *in-place* debugging.

Figure 3.11 shows a screenshot of the Anabranch debugger. The top row shows all the currently deployed streams. Clicking on a stream loads its details into the panes below. The left-most pane shows the stream's DAG, and the dependencies between them. A green vertex means the operator completed, and a blue vertex means the operator is running. Each vertex is labeled with its type (e.g., `map`), and the edges are dynamically updated with the last emitted domain values. The middle pane shows information about the selected vertex. The right-most pane shows a log of the received domain values and the emitted domain values that are computed based on the received value. Finally, the bottom pane shows the current state of the operator, which can be modified. The state of the transform operator shown in the screenshot contains the current value ("accumulator"), and the combiner function. The function is shown as `#Function<...>`, because Elixir cannot show an anonymous function its original code. A new state can be defined, e.g., a new function and accumulator, and deployed on the operator by pressing "Update". The text area at the bottom contains the arguments with which the operator was instantiated.

The meta-DAG in Creek has access to three pieces of information that are of interest for a debugger. First, the meta-DAG in Creek handles the events that arrive at an operator. These events represent the domain values, errors, and complete events. The meta-DAG can inform the debugger in real-time what internal messages are propagating through the stream. Second, the meta-DAG can intercept the base-level evaluation of these events and the result. This allows the debugger to show which datum is processed, and which datum is emitted as a result. Third, each meta-event contains information about the operator, including its state, arguments, and meta state. This can allow the debugger to inspect the current state of the operator, its type, and its initial arguments. Finally, the meta-DAG can modify the operator its state at runtime. This allows the debugger to send meta-level messages to a stream to modify its behavior at runtime.

Most meta-DAGs are closed DAGs with 1 source and 1 sink. The source and sink are provided by the runtime. The source emits the meta-events, and the sink consumes the meta-responses. The debugger is an external process that needs to communicate with a meta-stream, which is impossible. Ad-hoc communication is possible by sending messages to other actors, but this would be poor stream design; side effects have to be migrated to sources and sinks as much as possible. To solve this problem we adapted the meta-streams to allow an optional user-defined sink and source parameter. These user-defined source and sink can be used to emit or consume meta-events and meta-responses outside of the runtime. The debugger uses this mechanism to inject meta-events from the front-end into the meta-DAG, and to extract meta-responses to the debugger front-end. Figure 3.12 shows the graphical depiction of such a DAG. The *external source* is a source actor that can inject events directly into the meta-DAG. The *external sink* is a sink that can be used to extract data from the meta-DAG. In the graph shown in fig. 3.12 the source and sink are at the start and end of the stream, but they can be placed anywhere in the stream, depending on the application.

Anabranch shows that the meta-level can express complex meta-level applications, such as debuggers. However, the meta-level has a few shortcomings that limit the debugger its capabilities. First, the meta-level architecture lacks control over the stream as a whole. Intercepting a message and blocking the execution of the meta-stream is possible, but limited to a single operator. Blocking the stream as a whole requires control over multiple concurrent Elixir processes. The meta-level is a stream that is deployed on each operator separately, and can only communicate with the meta-DAG of its direct up- and downstream



Node: #PID<0.465.0>

Process: #PID<0.465.0>

Name: transform

Arity: 1:1

```
18:02:32 :: 1   -> 1
18:02:32 :: 10  -> 10
18:02:33 :: 2   -> 20
18:02:33 :: 11  -> 220
18:02:34 :: 3   -> 660
18:02:34 :: 12  -> 7920
18:02:35 :: 13  -> 102960
18:02:36 :: 14  -> 1441440
18:02:37 :: 15  -> 21621600
18:02:38 :: 16  -> 345945600
18:02:39 :: 17  -> 5881075200
18:02:40 :: 18  -> 105859353600
18:02:41 :: 19  -> 2011327718400
18:02:42 :: 20  -> 40226554368000
18:02:43 :: 21  -> 844757641728000
18:02:44 :: 22  -> 18584668118016000
18:02:45 :: 23  -> 427447366714368000
18:02:46 :: 24  -> 10258736801144832000
```

State:

```
{10258736801144832000, #Function<1.6575912/2 in Creek.DebugExample.test/1>}
```

Arguments:

```
{1, #Function<1.6575912/2 in Creek.DebugExample.test/1>}
```

Figure 3.11: *The Anabranch Debugger*

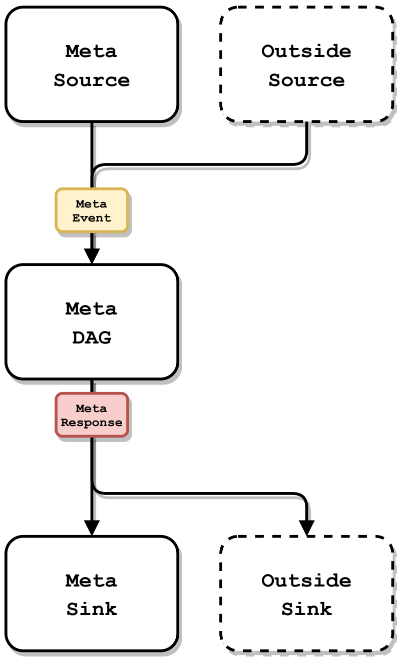


Figure 3.12: *The external source and sink for the meta-DAG. The dotted lines indicate that they are optional.*

operators. Second, the DAG that is deployed can be structurally changed with compile-time DAGs. This original representation is lost at runtime, and the debugger will therefore show the optimized version, which might hinder comprehensibility for the programmer. Finally, the debugger is a meta-level DAG, and the meta-level of Creek does not compose with other meta-level DAGs. This means that a DAG can only be debugged without any other meta-level DAGs.

Implementation Most of the complexity of the debugger is contained in the front-end JavaScript application. The meta-DAG essentially communicates bidirectionally with the front-end. The meta-DAG for source operators is shown in listing 29. The dag definition has an additional parameter, `sock`, which is the external sink. Every `tick` event is handled by calling the base-level. The result of the base-level is sent via the `sock` sink to the debugger. The debugger then displays that the source has emitted a datum. Additionally, the effects are executed, and the meta-response is sent to the meta sink `snk` (i.e., default behavior). The optional source is not defined in the DAG by the user, because it must be intercepted by the meta-source to wrap it with the meta data of a meta-event (i.e., operator, downstreams, and upstreams). A meta-event always contains meta-data of the operator it is about. Any external event is sent to the meta-source, allowing it to add the meta-data of the operator. The modified meta-message is emitted by the meta-source. The full meta-DAG definition for the debugger can be found in appendix G.

```
defdag source(src, snk, sock) do
  let tick? as filter(&match?({_, :tick}, &1))
  let rest? as filter(&(not match?({_, :tick}, &1)))

  let rests(as rest? ~> default ~> snk)

  let export as map(fn e ->
                  case e do
                    {p, {state, :tick, value}} ->
                      {:outgoing, p.pid, state, value}

                    {p, {_, :complete}} ->
                      {:complete, p.pid}

                    _ ->
                      :skip
                  end
                end)
                ~> sock

  let ticks as tick?
                ~> base
                ~> dup
                ~> (export ||| effects ~> snk)

  src
  ~> dup
  ~> (ticks ||| rests)
end
```

Listing 29: *The debugger meta-DAG for source operators.*

3.6.2 Performance Benchmarks

To evaluate the performance of the meta-level architecture we benchmarked an “identity meta” (listing 30) and compared it to a version of μCreek in which all meta-level machinery was removed, called Creek--

The `IdentityMeta` DAG (listing 30) is a meta-level program that does not modify the semantics of the base-level behavior. It does so by calling the base level (lines 6, 13, and 20) and executing the effects (lines 7, 14, and 21). This ensures that any overhead measured is introduced solely by μCreek_R.

For performance reasons the identity meta is optimized by applying operator fusion.

```
defmodule IdentityMeta do
    structure Merge

    defdag operator(src, snk) do
        src
        ~> base
        ~> effects
        ~> snk
    end

    defdag source(src, snk) do
        src
        ~> base
        ~> effects
        ~> snk
    end

    defdag sink(src, snk) do
        src
        ~> base
        ~> effects
        ~> snk
    end
end
```

Listing 30: *Identity Meta behavior*

To measure the performance overhead we ran two benchmarks for *μ*Creek as well as Creek--. The findings are reported below.

Fixed Amount of Values, Varying DAG Size Figure 3.13 shows the chart for the first benchmark. In this benchmark the amount of values propagated is fixed, and the amount of operators in the DAG varies from 0 (only sink and source) to 2000.

We observe that the execution time for *μ*Creek (blue) and Creek-- (orange) are both polynomial, but that the performance of *μ*Creek is slower by a factor of 3.3. In other words, the performance of the DAG decreases linearly with respect to the size of the DAG.

Fixed DAG Size, Varying Amount of Values Figure 3.14 shows the chart for the second benchmark. In this benchmark the number of values propagated varies from 0 to 10000, and the amount of operators in the DAG is fixed at 250.

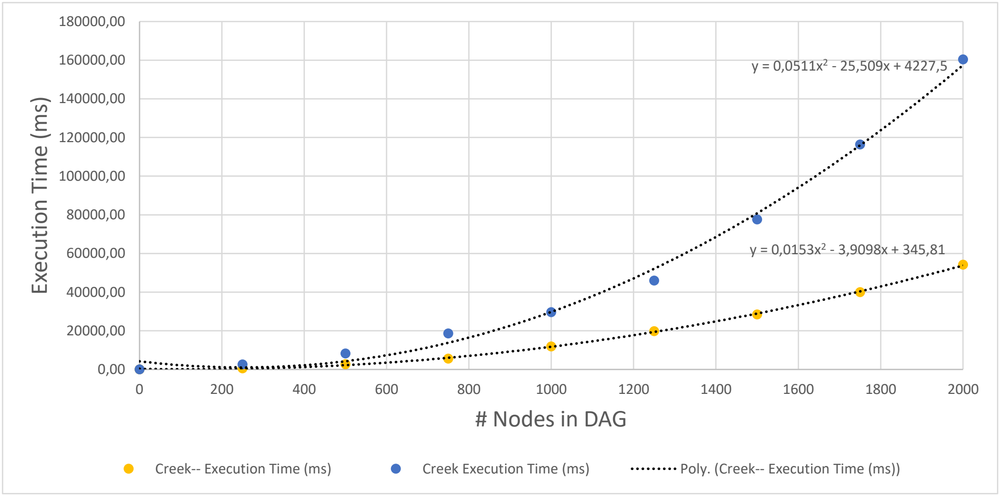


Figure 3.13: *Execution time in miliseconds with varying DAG size and fixed load.*

We observe that the execution time for μCreek (orange) and Creek-- (blue) are both linear with respect to the size of the input, but that the performance of μCreek is slower by a factor of 77.

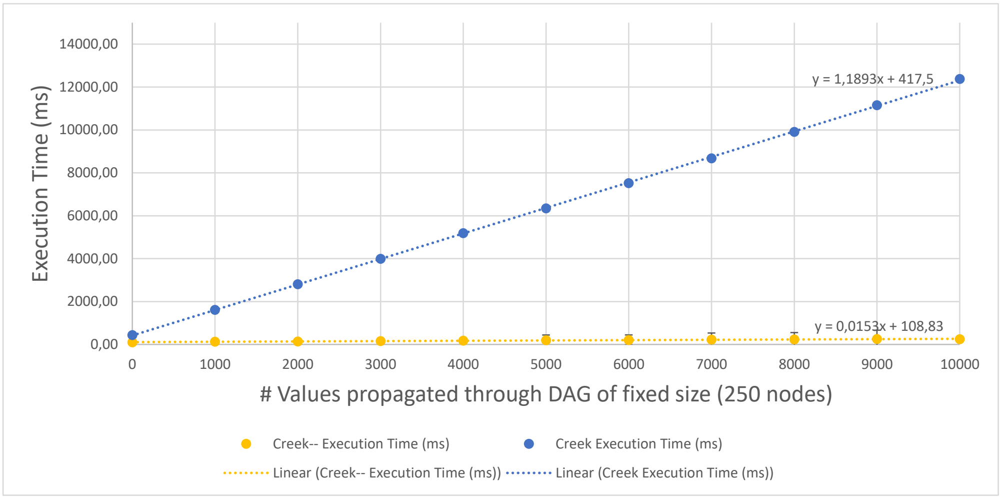


Figure 3.14: *Execution time in miliseconds with varying load and fixed DAG size.*

3.7 A Note on Related Work

In this section we discuss meta-level architectures in communication-based paradigms. μCreek is a meta-level architecture designed specifically for stream-based languages. To the best of our knowledge it is the first of its kind. We therefore broaden the scope of related work to include object-oriented, channel-based, and actor-based paradigms.

Reflection in Distributed Systems Garf [48] is an object-oriented distributed system that separates the behavioral aspects of an application from the functional aspects. The meta-behavior of the applications are contained within *encapsulator* and *mailer* objects. The encapsulator objects wrap base-level objects and can override the send and receive semantics of an object. CodA [70] is a meta-level architecture that was designed to express object-oriented semantics *bottom up*. Instead of reifying the behavior of an object-oriented language into a meta-level (i.e., *top down*), CodA provides seven meta-objects that can be composed to create many different base-level behaviors.

Both the above approaches provide the necessary mechanisms to express non-functional concerns such as asynchronous messages, distributed messages, and replication. By allowing the meta-level to intercept communication between objects it becomes possible to change the semantics of how messages are processed. However, both meta-levels are designed for object-oriented paradigms, while μCreek is a meta-level architecture specifically designed for stream-based languages. The architecture of CodA decomposes the meta-behavior into an extensible set of seven distinct objects that can each define their own semantics. In μCreek$_R$, the meta-level logic is divided into two conceptual phases that have no clear separation (i.e., *base* and *effects*), hindering the composition of meta-level behaviors.

In [14], the authors propose *channel reification*, where the communication between two objects is reified as a *channel*. Similar to CodA and Garf, a channel traps the communication between two objects at runtime and allows the meta-level to change its semantics. In the channel reification model, channels can be reused between objects if they have the same semantics, sender, and receiver. Conversely, in μCreek, every operator has its own instance of a meta-stream, which introduces significant overhead. As an extension to the channel reification model, [25] adds support for *multi-channels*, communication between multiple parties.

The channel reification model makes it possible for a meta-level approach to intercept messages while having access to the communication as a whole. A reified message in a channel contains the sender, receiver, and message. This approach results in a *global view*, and allows the meta-level to incorporate properties of the sender and receiver in its logic. However, in the context of a stream DSL, any operator should never be aware of the identity or properties of its up- and downstream operators and avoids coupling between operators. This invariant ensures that any operator can be composed with any other operator. The only identity an operator has of its up- and downstream is the gate to which it is connected. This is necessary to differentiate between streams in the base-level logic. For example, the `zip` operator needs this information to know when to emit a new tuple.

Reflection in Actor Langauges Akin to object-oriented languages offering meta-objects, some actor languages offer a meta-actor [90]. The meta-actor is responsible for intercepting incoming and outgoing messages of the lower-level actor it is tied to. A meta-actor captures the messages intended for the lower-level actor, as well as the messages the lower-level actor emits. Like `invoke` in object-oriented programming, and the canonical stream protocol in *μ*Creek, the most basic meta-level operators in the meta-actor are `send` and `receive`.

AmbientTalk [34] is an actor language that implements a mirror-based architecture based on the work by Bracha [20]. This meta-level architecture has many applications, such as adding proxy objects to the base language, persistence, and so on. In AmbientTalk intercession is used to build a "language laboratory", a programming language that can be easily extended.

The biggest difference between *μ*Creek and the work discussed above is that *μ*Creek is a meta-level paradigm that reifies the canonical stream protocol. In the work discussed above the meta-level works at the level of `invoke`, the most basic meta-level operation in an object-oriented language that executes an object its method. In context of stream programming, the most basic meta-level operation is the canonical stream protocol, i.e., `next`, `error`, and `complete`. In this regard, *μ*Creek is a more specific meta-level architecture than the generic ones discussed above. It is well-established that the goal of a meta-level should not reify too much [68], and in that regard, *μ*Creek can be considered a DSL for meta-programming streams, rather than a general-purpose meta-level architecture.

3.8 Summary of Chapter

*μ*Creek is a design for a metal-level architecture for stream-based languages. It consists of a compile-time part to transform the structure of a stream, and run-time part to modify the behavior of the stream execution semantics. We show that the design is expressive enough to separate the non-functional concerns from a stream in a structured way. Meta programming is popular technique to increase the extensibility and applicability of programming languages. However, to the best of our knowledge *μ*Creek is the first meta-level architecture for stream-based languages. *μ*Creek has been shown to be expressive, but the meta-level incurs a significant performance impact (77x slower). Further research is required to determine how to compose different runtime Directed Acyclic Graphs (DAGs) into a single meta-level behavior. Additionally, further research should investigate how to facilitate dynamic changes to a stream topology (i.e., higher-order streams).

Chapter 4

Potato: A Streaming Platform for CPS

A Cyber-Physical System (CPS) integrates ordinary physical devices (e.g., lockers on a university campus) into a unified digital system by equipping them with small computers. A network connection between devices and real-time data exchange can make ordinary physical systems more efficient. CPSs have found their way into a wide array of domains such as smart grids [5], smart buildings [50], medical systems [35], oil pipeline monitoring [13], and traffic control systems [53]. Despite technological advances in both hard- and software, CPSs are still challenging to develop, monitor, and maintain.

Chapter 2 introduced Creek, a stream Domain Specific Language (DSL) with a meta-level architecture that can be used to express non-functional concerns in streams. In this chapter, we use that as the foundation for a novel stream-based CPS architecture called Potato. In section 4.1 we detail an example of a CPS application. In section 4.2 we introduce the design challenges in terms of software engineering for CPSs raised by the literature, and in section 4.3 we define our approach to tackle these challenges. Section 4.4 shows how the basic building blocks of our design can be composed to build applications. Section 4.5 explains how Creek and μCreek (see chapter 2) can be used to cater to the non-function concerns of those applications. We conclude with a discussion of related work in section 4.6 and qualitative evaluation of our design in section 4.7.

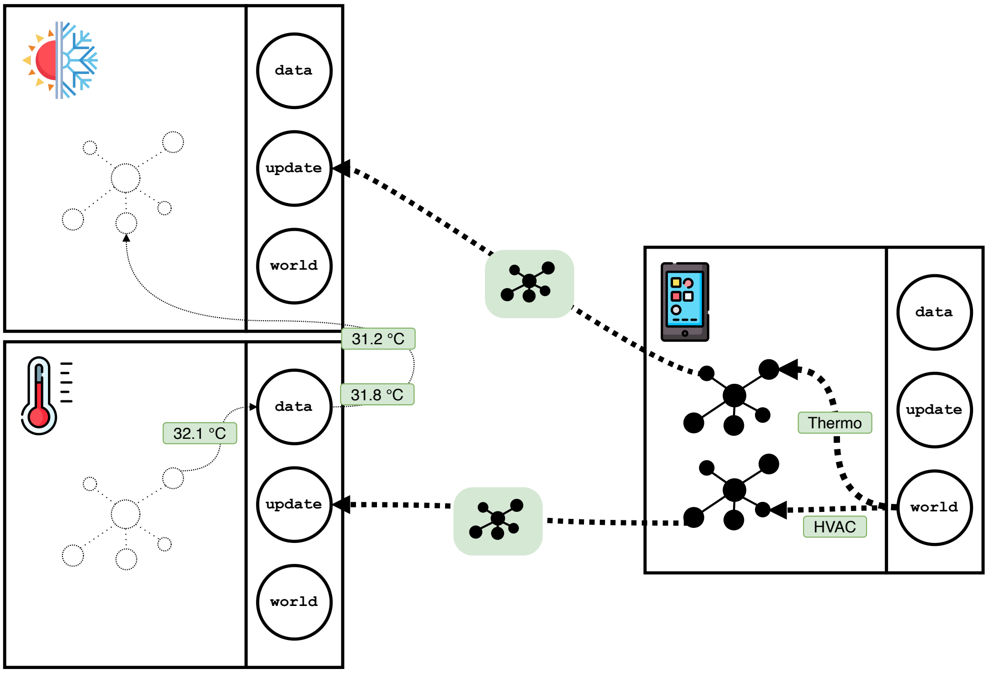


Figure 4.1: *An example of a network in* Potato. *A thermometer and HVAC system are programmed by a smartphone. The thermometer streams its measurements to the HVAC system directly, which turns itself on or off, depending on the temperature.*

4.1 Motivating Scenario: VUB 4.0 Campus

Before discussing the problem statement and our approach, we introduce a scenario that will serve as the basis of our validation in section 4.7.

The rector of the Vrije Universiteit Brussel (VUB) has a vision for a new campus: "VUB 4.0". The campus will be a testbed for new technology and will be equipped with state-of-the-art hardware. The campus has 50 lecture halls, with an average capacity of 120 students each, totaling 6000 students at full capacity. Each lecture hall is equipped with the following sensors and actuators.

- Thermometers, hygrometers, and carbon monoxide sensors to measure the air quality.
- Climate control to regulate the temperature in the lecture hall.
- A ventilation system to refresh the air.
- Dimmable lights.
- Motion sensors.
- Automatic windows that can be remotely controlled.
- Smart plugs at every seat.
- Access control panels at each door.
- Fire detection system.
- Alarm system speakers.
- A Wi-Fi access point that tracks the connected devices.
- Lockers for students.

The students and VUB personnel interact with the system using the VUB App on their smartphone. Students can use the app to access public lockers, to consult class schedules, and to receive notifications of absent professors. The VUB personnel can unlock doors, set ambient settings (i.e., temperature, draft, and lighting) in auditoriums, and infrastructure configuration (i.e., beamer input source and room computer preferences). Finally, the campus houses a handful of servers for telemetry, data processing, analytics, web hosting, and authentication.

The "VUB 4.0" system comprises thousands of devices. When the campus runs at full capacity, over 6000 phones are connected, totaling more than 10000 devices. In what follows, we discuss three scenarios that make use of the devices present on the VUB campus.

4.1.1 Scenarios

Scenario #1 Adaptive Lecture Hall Professor Remus is a computer science lecturer at the VUB. Every Tuesday, he teaches "Algorithms and Data Structures" to the first bachelor students in the Lambda auditorium from 10 to 12. The locks use the teaching schedule to determine who can unlock the auditorium. When Professor Remus nears the door at the beginning of his lectures, the door unlocks automatically. When Professor Remus enters the auditorium, the temperature is set to 20 degrees, and the ventilation is set to 20%. Because Professor Remus brings his own computer, the room computer is turned off, and the beamer automatically switches from HDMI to USB-C input. The lights in the front row are dimmed to increase the visibility of the slideshow. The rest of the lights are turned on to full brightness. Remus wants his students to pay attention, so the smart plugs for all students are turned off, and the Wi-Fi access point only allows Professor Remus' laptop to connect to the internet.

Scenario #2 Noisy Campus Some parts of the campus are still under construction. Trucks are entering and leaving, and construction workers operate loud and heavy machinery during lecture hours. The math professor, Professor Boole, cannot bear having her lectures disrupted by construction work noises, so she writes a stern note to the VUB infrastructure office. To avoid another argument with the professor, the VUB allows lecturers to override the settings of the automatic windows. The IT department reuses old sound level meters and hangs them around the campus. The VUB app of the lecturers is updated over the air with new functionality: windows are closed automatically if the noise outside exceeds a certain threshold. Professor Boole immediately toggles this functionality so that each auditorium she teaches in closes its windows automatically if the measured noise outside exceeds 50 dB.

Scenario #3 Student Tracking Profesor Theodore, another computer science teacher at the VUB, teaches Programming Language Design. Professor Theodore is passionate about his course and hopes his students take away as much as possible from his lectures. However, some of his students have classes on the other side of campus and might be a little late for class. Professor Theodore decides to poll all his students through the VUB application, asking if they are on their way. Students who are not present in the auditorium but who are on campus receive an instant poll from the VUB app. A student can either ignore the professor or tell him they're on their way. Professor Theodore waits a few minutes for responses and then begins his lecture.

4.1.2 Conclusion

The "VUB 4.0" campus is a contemporary example of a CPS in terms of software engineering as described in the literature. It features heterogeneous devices [81, 72, 10, 91], limited capability devices [81, 72, 10], mobility [18, 72, 10], distributed data processing [81, 72, 10], implicit interactions [18, 81], dynamic code updates [55, 21, 81], unreliable networks [18, 72, 81, 10, 91], and is large scale [81, 72, 91]. Additionally, smart buildings are a use case for validation in many related works [73, 87, 26, 12].

The three scenarios that we distilled from the "VUB 4.0" application will be used in this chapter to show how our approach can be used to express the data streams in a CPS. Additionally, at the end of this chapter we implement the three scenarios in Potato and how our approach addresses the challenges.

4.2 Design Challenges in Cyber-Physical Systems

Designing, deploying, and maintaining Cyber Physical Systems (CPSs) is an undertaking that requires expertise in hardware as well as software. This dissertation focuses on the software development aspect of CPSs, in particular the programming language aspects of designing CPSs. In this section, we look at the inherent properties of CPS and the existing approaches to deal with these traits.

Large Scale CPSs come in *different scales* [85, 5]. Smaller CPSs, such as smart home systems, consist of a few dozen devices [12], while systems like pipeline monitoring [13] and smart grids [5] consist of thousands of devices. The scale of these systems affects the design of software for these systems. Without the right abstractions, developers are burdened with defining and maintaining pairwise connections between a large number of devices.

The "VUB 4.0" campus contains thousands of devices. When all students are present, there are 6000 smartphones connected. Depending on a device its role in the application, it may require different data streams. *It is impossible to predefine all connections in the application, so they have to be defined dynamically. Suitable abstractions are necessary to make this possible for applications of any size.*

Open System CPSs are *open systems* [53, 10, 18]. An open network allows devices to join and leave the application when they want. This behavior is due to mobile devices, such as smartphones, but also because devices being replaced while the system is operational. As a consequence, the CPS has to be able to integrate devices at runtime, without knowing them beforehand. Manually provisioning devices before connecting them is impractical because cyber-physical systems are not typically taken offline to apply updates. The CPS applications need to be able to automatically provision and adapt to new devices in the network.

The "VUB 4.0" campus contains a variety of sensors and smart devices scattered across the campus. Devices such as motion sensors and thermometers are cheaper to replace than they are to repair. When a sensor is replaced, the part of the application that relied on its `data stream` must adapt to the new sensor. To achieve this behavior, a form of service discovery and dynamic update is required. *Service discovery is helpful to detect new devices on the network, and dynamic updates are helpful to update new devices with the application logic to integrate them at runtime.*

Unreliable Network and Devices Devices in a CPS are *inherently unreliable* [18, 72, 81, 10, 91]. Devices in the system can be battery-powered or connected over unreliable connections. The high probability of failure entails that no guarantees can be offered about a device's availability or the services it offers. These frequent disconnects impact the reliability of data exchange and make the use connection-oriented approaches such as sockets, RPC, and message passing that rely on physical addresses difficult. Using connection-oriented approaches would pollute the application code with exception handling and logic to replace the lost device's service. Because failure is not an exceptional state in CPS, connection-oriented paradigms are unfit.

The VUB smart campus has deployed multiple battery-powered thermometers in every lecture hall to control the HVAC system. When one of these sensors fails, the HVAC system has to communicate with a backup sensor, and the faulty sensor must be replaced. *Devices should be addressed based on their properties (e.g., location) and capabilities (e.g., thermometers), called intentional addressing, rather than their physical address, called extensional addressing*

Heterogeneous Devices The *devices in a CPS exhibit heterogeneous traits* [81, 72, 10, 91] in terms of their computational power, data encoding, and communication protocols. These differences stem from multivendor products, different non-functional concerns (e.g., Kelvin and Celcius thermometers), and the heterogeneous physical devices they represent. These differences are the most problematic when integrating devices that are functionally equivalent (e.g., thermometers), but differ in how they provide this data. These differences can be classified as *non-functional* differences. When integrating heterogeneous devices, the applications must deal with the non-functional differences using device-specific integration logic. Entangling these concerns is a bad software engineering practice and should be avoided.

In the "VUB 4.0" campus, the sound level meters are gathered from all over the campus, and the devices are from a variety of manufacturers. Some devices are plugged into a power outlet and continuously stream their measurements over the network. Other types are battery-powered and only measure the current level on-demand. Another type of meter emits data encoded in JSON format, while most meters emit data in plain text format. Integrating just different types of sensors leads to an increase in complexity in the application logic. *The heterogeneity of the devices should be addressed separately from the application logic to avoid polluting the application logic with glue code.*

Data-Driven A CPS is a perpetual feedback loop of data and instructions between the devices in the system [81, 72, 10]. Each device relies on another device to function and is relied upon by other devices in turn. A single event can start a chain reaction of events throughout the system. Expressing event-driven systems is difficult in traditional sequential paradigms [16]. Callbacks – parts of the application logic – are executed when a specific event occurs. Callbacks make a program structurally difficult to understand, and are difficult to coordinate [16].

It should be evident from the three scenarios in section 4.1 that the entire VUB campus is an data-driven system. Scenario 1 shows that moving around on campus creates data streams between devices. Scenario 2 is centered around data streams coming from the sound meters. Scenario 3 shows that temporary streams can be set up to gather data from the real world. *CPSs need programming abstractions that help in defining data streams between devices, regardless of the scale of the network.*

Limited Computational Power Some devices in a CPS may be limited in their computational power relative to the computations done in the system [81, 72, 10]. The limited computational power impacts the types of services a device can offer. Devices may have to involve other devices to provide a computationally intense service. A well-known approach to this problem consists of offloading computations to other devices in the network [31, 39, 99]. However, these approaches are often not integrated into the paradigm of the rest of the application and require manual code management of the developers.

Most of the "VUB 4.0" campus consists of devices with limited computational power, e.g., thermostats, hygrometers, motion sensors, and smart plugs. These devices may involve other devices to provide their services. For example, a thermostat deploys its application logic to control the HVAC system on the

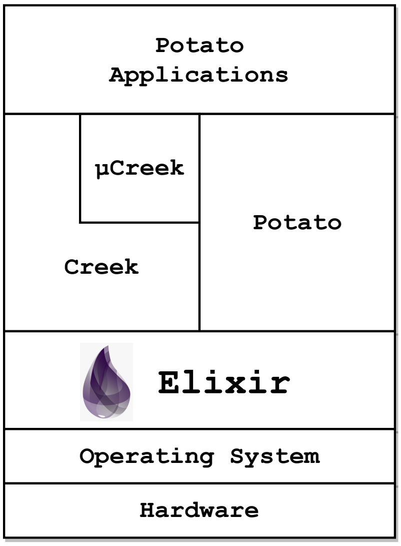


Figure 4.2: *The technology stack used to implement* Creek *and* Potato *in Elixir.*

HVAC system itself because it has more computational power. *The limited capabilities of some devices impact the application logic they can execute. The ability to involve other devices in a computation can mitigate these limitations.*

4.3 Potato: A Stream-Based CPS Framework

In the previous sections we introduced the design challenges in Cyber-Physical System (CPS) design. In this section we present and discuss our approach, called Potato, and discuss its basic building blocks, and how they relate to the design challenges from section 4.2.

Central to Potato is the stream programming paradigm. The most important concepts from a CPS are the network, the devices in the network, the data streams between devices, the application level programs, and the non-functional concerns. These core concepts of a CPS can be represented by a stream, allowing the entire system to be expressed in a single unified paradigm. Representing these concepts as a stream of data guides the programmer to define the application logic as a declarative, functional, stateless transformation of these streams. Making streams a first-class citizen allows the application to pass them around to setup distributed services.

In what follows, we present and discuss the basic building blocks of a Potato application, and how they relate to the design challenges. Potato represents the most central concepts of a CPS in a unified paradigm, stream programming.

Before explaining them in detail, we give a brief overview. *Each device in a* Potato *application offers the following concepts.*

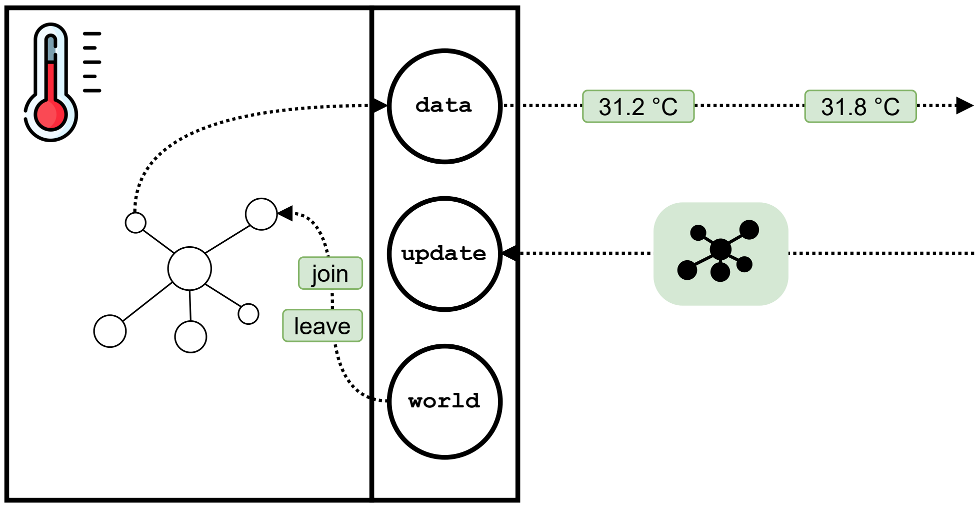


Figure 4.3: *A graphical depiction of a* Potato *node representing a thermometer. The world-data- and update stream are shown on the left. The* `world stream` *provides data to the locally deployed streams, which in turn multicast data on the* `data stream`.

- **Node Descriptor** Each device in a Potato network is described by a *node descriptor*, a data structure containing meta information about the device such as its name, type, and manufacturer. When the device connects to the network it broadcasts its node descriptor to other devices to announce its presence. The node descriptor forms the basis for service discovery.

- **World Stream** The built-in `world stream` emits network changes to the application, e.g., device joins and leaves. The `world stream` emits an event when a device leaves or joins the network along with its node descriptor.

- **First-Class Reactive Programs** First-class reactive programs are programs that can be transmitted across the network to execute elsewhere, and make it possible to create distributed streams of data.

- **Data Stream** A device in a Potato application has a dedicated `data stream` that can be used to multicast data onto the network. Other devices in the application can observe this `data stream` if they are interested in the data of a particular device.

- **Update Stream** The Potato framework observes a *stream of first-class reactive programs*, called the `update stream`. Any device in a Potato application can stream applications to another device, which will deploy them locally. First-class reactive programs and the `update stream` combined make it possible to create distributed applications.

Figure 4.3 graphically shows how these streams are used. The diagram represents a thermometer device. The `world stream` emits join and leave events to locally deployed streams. The locally deployed streams in turn stream data to the network via their data stream. The `update stream` consumes incoming first-class reactive programs, and deploys them locally.

Hello, Potato! As a first example of a Potato application, consider the listing in listing 31. The application is deployed on Professor Remus his laptop, and turns off all the heating systems in the same room as his laptop. Of course the semantics of the snippet will not be entirely clear until the end of this section, and the application only serves as a first introduction to Potato.

The application defines a description of the local device, i.e., the *node descriptor* (line 3). announces itself on the network (line 4), and then deploys an instance of the `heating` stream locally with the `world_stream` as its source. The `heating` stream filters out all join events of `heating` devices. For each of these devices, a first-class program `p` is created (lines 16–19), and deployed on the heating device (line 19).

In what follows we discuss these concepts individually and explain how they can address the challenges from section 4.2.

4.3.1 Node Descriptor: Device Identification

Every device in a Potato application is represented by a *node descriptor*. A node descriptor contains dynamic and static properties of a Potato device. Every node descriptor uniquely describes a single device in an application. Listing 32 shows a node descriptor for an Android device, called "Boole‘s Phone".

A minimal node descriptor should at least contain a unique identifier `id`. Other values are application dependent. By convention, the identifier, hardware, and alias of a device are *static*. Other fields in the node descriptor can be changed at runtime by the node, such as `location`.

A node descriptor is *local* if it represents the underlying Potato system, and is *remote* if it represents another Potato instance. A device can modify its local node descriptor to reflect changes in the device's physical properties. When a device changes its node descriptor it is broadcast again. Any device observing the network will receive the node descriptor with the updated values.

The node descriptor must be installed together with the Potato runtime on the device before it is deployed and connected to to the network. When the device

```
Laptop
defmodule Laptop do
  def main(room) do
    nd = %{hardware: :laptop, name: "Remus's MacBook"}
    Potato.Network.Meta.set_local_nd(nd)
    deploy(heating, world_stream: Potato.world_stream,
                    snk: Creek.ignore(),
                    room: "Mandela Auditorium")
  end

  defdag heating(world_stream, snk, room) do
  world_stream
    ~> filter(fn {event, device} ->
      event == :join and device.hardware == :heating and device.room != room
    end)
    ~> map(fn {_, device} ->
      p = program do
            Heating.turn_off()
          end
      Subject.next(device.update_stream, p)
    end)
    ~> snk
  end
end
```

Listing 31: *A* Potato *application that turns of all the heating systems in the same room as the laptop.*

powered on, and connects, or reconnects, to the network, it announces itself by broadcasting its node descriptor. Every device on the network will receive an event on their `world stream`.

4.3.2 World Stream: Service Discovery

A CPS is inherently an open system. This entails that new devices can join the application over the course of its lifespan. Additionally, devices connect and disconnect at undefined moments due to their volatile connections. To deal with these challenges, Potato offers the `world stream`, a built-in stream that emits network events whenever changes on the network occur. The streams are a declarative approach to address devices in a scalable way.

A network event is structured as a tuple that consists of the type of event (i.e., `join`, or `leave`), the node descriptor of the device in question, and a reference to their `update stream` and `data stream`. Listing 33 shows an example network event emitted by a `world stream`. The event contains the description of the device that joined (i.e., the professor's smartphone), a `join` tag to indicate the phone just connected to the network, and references to the device's data and

```
%{
  hardware: :android,
  type: :phone,
  alias: "Boole's Phone",
  id: "75615454-9968-4e15-832c-120a726d5875"
  location: nil
}
```

Listing 32: *Node descriptor for Professor Boole's smartphone.*

```
{:join,
 %{
   hardware: :android,
   type: :phone,
   alias: "Boole's Phone",
   id: "75615454-9968-4e15-832c-120a726d5875"
   location: nil
   data_stream: #<Stream 0.451>,
   update_stream: #<Stream 0.394>
}}
```

Listing 33: *Example of a datum emitted by the `world` stream.*

`update stream`.

The Potato runtime continuously monitors the network and emits the event as soon as a change occurs. This ensures that every Potato device has an up-to-date snapshot of the current state of the network at all times. When a Potato device connects to the network, the network is scanned for currently connected devices. For each device the `world stream` emits a join event, to reflect the current state of the network.

```
defdag heating(world_stream, snk, room) do
world_stream
  ~> filter(fn {event, device} ->
    event == :join and device.hardware == :heating and device.room != room
  end)
  #...
end
```

Listing 34: *Example of designation using the `world stream` in Potato.*

Listing 34 shows the designation part of the Directed Acyclic Graph (DAG) from listing 31. The `heating` DAG uses the `world stream` as its source, and filters out all devices that are not of type `:heating`, and in a different room. The `world stream` can be transformed using all Creek operators, The `world stream` is a so-called *cold observable*, in Reactive Extensions terminology. When a DAG is deployed that uses it as a source, the `world stream` will emit all previously emitted values to the new stream. When a DAG is deployed that uses the `world stream` as its source, the `world stream` will first emit the current state of the network as series of `join` events. Join events followed by a leave event cancel each other out, and are removed. After these events have been emitted, the real-time changes are emitted.

To show the `world stream` its behavior, consider fig. 4.4. The x-axis presents time, and each white square represents a stream. The horizontal lines starting at a stream are the values emitted over time by that stream. In the example, stream `A` and `B` use the `world stream` as their source. An upward arrow indicates the point in time where stream A and B are deployed. At `t1`, stream `A` is deployed. Next, device `A` and `B` join, and the respective events are emitted on the `world stream`. Because stream `A` uses the `world stream` as its upstream, it also receives these events. At time `t2`, a new stream `B` is deployed. The cold observable behavior of the `world stream` causes the `world stream` to first emit a join event for device `A` and `B` to stream `B`.

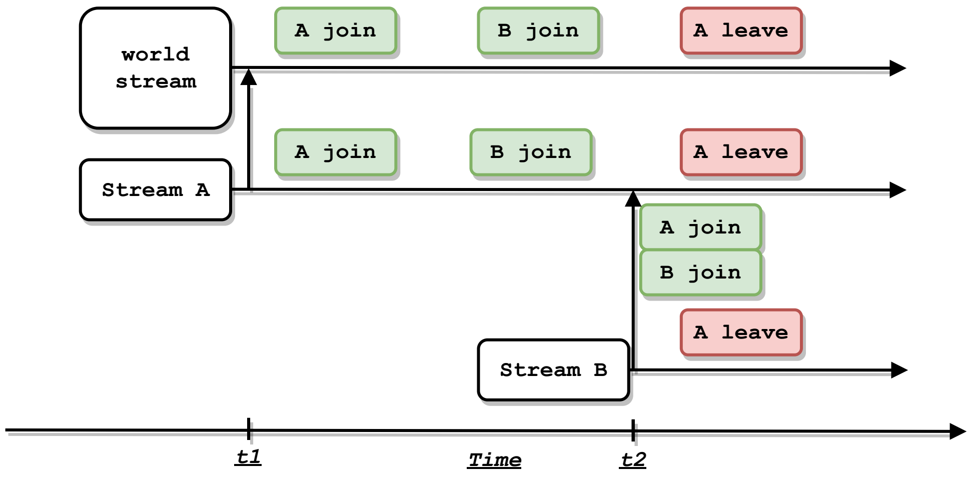


Figure 4.4: *The `world stream` is a hot observable. All previous values are emitted to each new stream that uses it as a source.*

A stream of network events results in **a declarative approach to creating subsets of the network (designation)**. Consider the case where one wants to create a stream of all devices of type `android`. Using the reactive streams operators, it becomes as trivial as applying a `filter` operator on the `world` stream to ignore all the events where the node descriptor does not contain an `android` value for the `hardware` field.

The stream paradigm combined with the `world stream` and node descriptor, forces applications to deal with a changing network by design. The `world stream` must be used as a stream source to obtain information about devices and the network. In doing so, the changing nature of the network is apparent, and cannot be avoided.

4.3.3 First-Class Reactive Programs: Dynamic Updates

Potato offers a form of code mobility to exchange instructions between Potato instances, and to offload computation to another instance. Code mobility allows computations or programs to be moved from one device to another to execute them remotely. To facilitate code mobility, reactive programs and streams are first-class citizens in Potato. Streams and programs can be assigned to variables, passed to functions, and sent over streams. A first-class reactive program is created using the `program..end` special form. A `program` expression creates a first-class value that represent a delayed computation.

Listing 35 shows part of an application that runs on a thermometer. The thermometer uses a first-class program to control an air conditioning system. On the thermometer a source that emits the current measurement is created (line 1). The first-class program (lines 3–14) defines (lines 4–12) and deploys (line 13) a DAG that consumes the stream of temperatures.

```
Thermometer
1  local_temps = Source.function(fn -> current_temp() end)
2
3  p = program do
4      d = dag react_to_temp(temperatures) do
5            temperatures
6            ~> each(fn t ->
7                 if t > 23.0 do
8                   HVAC.on()
9                 else
10                  HVAC.off()
11                end
12              end)
13     deploy(react_to_temp, [tempeartures: local_temps])
14   end
```

Listing 35: *A first-class program that sets the target temperature of the HVAC system to 24 degrees.*

Strictly speaking, a first-class Potato program can contain any valid Elixir code, including unbounded recursion. Allowing arbitrary code and unbounded recursion allows for programs to never terminate. In Potato, a *well-behaved program* can deploy streams and create sources and sinks, but should never block and only use bounded recursion. A well-behaved program deploys the necessary streams on the device and then terminates, leaving the streams deployed. Recall that distributed streams terminate if remote parts are disconnected (section 2.3), ensuring the stream is well-behaved in the face of network failures.

When a program is defined, it captures its lexical scope by copy. References to sources and sinks are also passed by copy, making it possible to set up transparently distributed streams. The combination of code mobility and remote references allows applications to create distributed streams of data. The previous example in listing 35 captures the `source` variable in its lexical scope. If that program is deployed on another device, a distributed stream is created. The `source` stream is located on the originating device, while the rest of the stream is running on the destination device.

Potato applications cannot explicitly evaluate a first-class program. Their purpose is to be transmitted over streams to other devices. Delaying computations should be done in the host language if possible. In Elixir, for example, a computation can be delayed using lambda abstractions.

In section 4.3.5, we explain the details of code mobility in a Potato application, and detail the evaluation mechanics of first-class programs.

4.3.4 Data Stream: Data Dissemination

CPSs require abstractions that remain expressive, regardless of the scale of the network. Defining pairwise dependencies between devices does not scale well beyond a few dozen devices. In this section, we introduce the `data stream`, a unique stream for each device that allows it to emit data to all its downstreams. Devices can use another device's `data stream` as the source for a stream.

Listing 36 serves as a minimal example on how to disseminate data on the network. The `emit_temperature` DAG emits a temperature measurement every second when it is deployed. The local `data stream`, `myself.data_stream`, is used as the sink. Every value is thus emitted to each device that is using the device's `data stream` as a source at that point.

A device obtains a reference to another device's `data stream` through a node descriptor in the `world stream`. The device can use the reference to the `data stream` as a source for other DAGs. It is the principal means of obtaining data from another device in Potato.

The `data stream` has a dual purpose in a Potato application. The device where the `data stream` originates, can use it as a sink. The other devices in the network can only use the `data stream` as a source. When a device emits

```
source = Source.range(0, :inifinity, 1000)                    Thermometer
sink = myself.data_stream

defdag emit_temperature(src, snk) do
  map(fn _ ->
    {:temp, Sensor.read()}
  end)
  ~> snk
end

deploy(emit_temperature, src: source, snk: sink)
```

Listing 36: *Reactive DAG to continuously read out a sensor and emit onto the network through the data stream. The local `data_stream` is used as a sink in the `emit_temperature` DAG.*

a value onto its `data stream`, it is *multicast* to all devices that have a stream deployed that uses it as a source.

When a device emits a value, it is sent to all the devices that are using the device its `data stream` as a source. Emitting each value to all these devices can induce unnecessary communication overhead, especially if the device is the source of many different data streams. Dedicated streams avoids sending unnecessary data over the network. Deploying first-class reactive programs on remote devices, and sending first-class streams as data to remote devices, allows applications to create dedicated streams. This avoids the communication overhead of emitting values over the `data stream` but adds accidental complexity to setting up the stream. In section 4.4.4 we show this approach in more detail.

4.3.5 Update Stream: Real-Time Updates

The open nature of CPSs requires devices to be provisioned at runtime. Additionally, due to heterogeneous devices and limited resources, devices need to involve other devices in distributed computations for performance reasons, which can also be done by updating the software at runtime.

Our approach for tackling these challenges is to allow devices to deploy short-lived programs, or *ephemeral updates*, onto other devices using *first-class programs*. This allows a Potato system to reconfigure itself and adapt to changing circumstances caused by the open network and volatile connections.

Listing 37 shows a minimal example of code deployment. The code is executing on device A. The code assumes `device_b` hold a node descriptor for device B. Device A defines a program `p` that, when executed, turns on an alarm. Program `p` is emitted onto device B its `update stream`, transmitting the program across the network and executing on device B. The result is that `"Turning on alarm!"` will be printed on the console of device B and that its alarm will be sounding.

```
device_b = ...

p = program do
  IO.puts "Turning on alarm!"
  turn_on_alarm()
end

Subject.next(device_b.update_stream, p)
```
Device A

Listing 37: *Device A deploying an update on a remote device, B.*

Similar to the `data stream`, all Potato devices have an `update stream`, which is a sink that can be used by other devices. Devices can obtain a reference to this stream through the node descriptor. Emitting a first-class program onto an `update stream` deploys the program on the owner of the `update stream`. This approach serves two purposes. First, a device can update another device to create ad-hoc services. Devices do not need to have the domain logic installed beforehand because it can be contained in an update. An example of ad-hoc service is updating a thermometer to make it broadcast measurements. Second, a device use updates to involve another device to offload a part of a stream.

A device has no control over its `update stream` from within the application logic. The application can only use the `update stream` of a device to send programs to other devices. Potato does not expose a primitive to manually deploy or evaluate programs, so the application cannot simulate the `update stream` behavior either. This limitation is by design. When an update is deployed onto a remote device, it is registered locally. This allows the Potato runtime to pause, store, or remove updates.

Well-behaved first-class programs deploy streams on the device they are executed on. In case of a network partition, the streams they deployed will be terminated by the stream termination protocol. If a program only deploys streams, it ensures that there program has no lasting effects on remote device in case of a network partition. Hence, a program is *ephemeral*. There is an exception, however. An ephemeral program that deploys a source, sink, and stream on the same device, cannot be affected by a network partition. The only way such a stream terminates is when the stream is finite, e.g., it contains `take(n)` or `first()` operators, or the source is finite.

Every program deployed on a device through the `update stream` must be executed concurrently with other deployed programs, allowing a device to compute multiple streams at once. Additionally, programs should be scheduled fairly, to avoid deadlocks. Consider the scenario where a source and a sink are on the same device. Unfair scheduling can lead to deadlocks in the stream. Programs cannot directly communicate with one another and are not aware of each other.

4.4 Building Applications with Potato

The previous section introduced the basic building blocks of a Potato application: node descriptors, first-class programs, the `world stream`, the `data stream`, and the `update stream`. This section shows patterns to use these building blocks to build applications that handle the scale, volatility, open nature, heterogeneity, and distributed computations of Cyber Physical Systems (CPSs).

4.4.1 Stream-Based Service Discovery

As we explained in chapter 1CPSs are built on top of dynamic networks with volatile connections. Devices in such a network cannot rely on static network infrastructure to discover services. In Potato, devices announce themselves on the network by broadcasting their node descriptor. The node descriptor contains the necessary information for other devices to determine whether it is relevant. Applications can discover devices by using the `world stream` as a source and transforming it with stream operators. A stream of available services can be created by transforming the `world stream` using declarative operators (e.g., `filter`).

The process of selecting one of the devices of interest is called "addressing" or "designation". Potato uses *logical addressing* [74], allowing devices to be designated based on logical properties (e.g., type or location). Other types of addressing, such as *physical addressing*, require a device to communicate with another device using physical addresses such as IP addresses or another unique identifier. Physical addressing does not scale beyond a few dozen devices and does not work in an open network.

Listing 38 is an example of the idiomatic approach in Potato to designating devices. The application prints out the id of each thermometer on the network. The first Directed Acyclic Graph (DAG), `thermometer_stream` filters out all events that are not `join` events (line 3) from its source. Next, each `thermometer` is filtered out (line 4). Finally, the event tag is stripped, and only the node descriptor is passed along (line 5). Any downstream of an instance of the `thermometer_stream` stream can assume that the values are thermometer node

```
1  defmodule PrintThermometers do
2    defdag thermometer_stream as
3        filter(fn {event, nd} -> event == :join end)
4        ~> filter(fn {:join, nd} -> nd.type == :thermometer end)
5        ~> map(fn {:join, nd} -> nd end)
6
7    defdag print_thermometers(world) do
8        world
9        ~> therometer_stream
10       ~> each(fn thermometer ->
11            IO.puts "Thermometer discovered: #{thermometer.id}"
12          end)
13   end
14 end
```

Listing 38: *Printing out the identifier of all the thermometers in the network.*

descriptors of devices that just joined the network. The `print_thermometers` DAG defined in lines 7–13 transforms the `world stream` using `thermometer_stream`, and prints out a message on the console each time a thermometer is discovered on the network.

The declarative stream-based approach to designation allows devices to monitor the network for changes and to discover devices *reactively*. As soon as a device connects to the network, the runtime emits an event on the `world stream`, triggering the recomputation of all the streams that depend on it.

4.4.2 Code Deployment & Offloading

Potato allows new devices to connect to a running distributed application. When a device connects for the first time, it will not have any application code deployed. Potato can be considered an "empty virtual machine." A device only has the required node descriptor, and code to talk to its physical device, i.e., *utility code*. In Potato, a device without application code is called a *blank device*. When a device A is discovered via a join event by another device B, B assumes that it has not yet deployed any code on A. This assumption can be made for two reasons. First of all, the idiomatic approach in Potato is to provision devices with utility code and to deploy application logic at runtime. Secondly, as soon as a distributed computation is interrupted, the stream termination protocol (see section 2.3) ensures that the streams are stopped and removed from the device.

Listing 39 shows the idiomatic Potato approach to obtain a single reading from all the thermometers in a network via a smartphone. The smartphone deploys a program on each thermometer that makes the thermometers emit a single

Smartphone

```elixir
defmodule AverageTemperature do
  defdag update(world, snk) do # Stream 1
    world
    ~> timeout(60)
    ~> thermometer_stream
    ~> each(fn thermometer ->
      p =
        program do
          d = dag emitter(src, snk) do
                src
                ~> take(1)
                ~> snk
              end
          src = Source.function(fn -> read_thermometer() end)
          deploy(d, src: src, snk: snk) # Stream 3
        end

      emit(thermometer.update_stream, p)
    end)
  end

  defdag gather_results(src, snk) do
  snk
    ~> window(60)
    ~> snk
  end

  def get_average_temp() do
    snk = Creek.subject()
    vls = Creek.Sink.first(self())

    deploy(gather_results, src: snk, snk: vls)    # Stream 2
    deploy(update, snk: snk, thermometers: world) # Stream 1

    receive do
      {:first, xs} ->
        IO.puts "Average temperature: #{sum(xs) / len(xs)}"
    end
  end
end
```

Listing 39: *Computing the average temperature per minute of all thermometers.*

reading. The smartphone listens for these readings, and calculates the average temperature. The application consists of two major DAGs, the update and gather_results DAGs.

The update DAG monitors the world stream, and deploys a first-class program onto each thermometer it discovers (lines 2–18). The first-class program deploys a stream on each thermometer that emits a single measurement into a sink (line 15) defined on the smartphone. The sink is *passed by reference* via the first-class program (line 15), creating a distributed stream between the smartphone and each thermometer. Passing a source or sink by reference means that the process itself is not moved, but a reference to the process that allows other streams to transparently communicate with it, as if it was running locally.

The `gather_results` DAG uses the smartphone sink as a source by observing it for a minute (line 24) and emitting the gathered values in another sink (line 25). An instance of the `gather_results` and `update` DAGs are deployed (lines 32-33).

The application in listing 39 shows how the deployment of ephemeral updates on devices can create a distributed stream. The `first` sink running on the smartphone will terminate as soon as it has received a list of measurements. This will terminate the entire upstream of the sink, including all the remote streams running on the thermometers. Streams can be distributed by capturing references to sources and sinks in the scope of a first-class reactive program. Together with the stream termination protocol the runtime can guarantee that the updates are ephemeral.

4.4.3 Handling Failures

The volatile nature of a CPSs requires applications to deal with failures in the network. The failure of a device in the network should not cause perturbations throughout the entire application. In Potato, we have chosen to adopt the Erlang mantra of "let it fail". Potato applications are designed with failure as a primary concern. Instead of trying to recover from each failure, the application lets the device disconnect and tries to continue working with other devices.

The operators of a stream can be distributed over multiple devices in the network. The connections between these devices can be disturbed at any time. We consider two types of network failures; short perturbations in the network that interrupt connections for a few seconds, and longer-term disconnects.

- **Short perturbations** imply that the device promptly reconnects, while longer-term disconnects imply the device will not reconnect. Short perturbations are allowed in a configurable grace period by Potato to avoid handling disconnects that are rapidly followed by reconnects.

- **Long-term disconnects** are presumed to be permanent, and streams that rely on that device are terminated (see section 2.3). This implies that

> the disconnected device can make the same assumption; as soon as it cannot connect to the application, it can terminate all running streams. This ensures that when a network partition is resolved, devices from both partitions treat each other as new devices.

The advantage of the "let it fail" approach is that devices are allowed to disconnect at any time and do not have to keep track of state regarding previous connections. Dealing with disconnects requires devices to rebind to other devices for an equivalent service. Because applications are written as a declarative transformation of the `world stream`, an equivalent device will be emitted once it connects to the network. The downside of this approach is that the ephemeral updates that constitute a service will be sent multiple times. If a given device connects and reconnects frequently, this incurs overhead in network transmissions. Additionally, the time between the disconnect and a rebind is time during which data cannot be transmitted.

4.4.4 Dedicated Streams

Potato devices may deploy ephemeral updates on other devices to create services on demand. Multiple services can run on a single device, and the consequence of multiple services is that the data of each service will be multicast to all other devices as well because the `data stream` is a shared stream. Consider the scenario where multiple devices listen to the same device, each for different data streams. The `data stream` always multicasts to observing devices, causing both consumers to receive all data. In other words, consumers have to explicitly filter out their data. The `data stream` is elegant to express simple services, but induces unnecessary data transmission across the network when scaling up to multiple devices.

Dedicated streams resolve these issues by only sending data to select devices. The idiomatic approach to create dedicated streams in Potato is by leveraging the pass-by-reference semantics of sources and sinks. When the program is deployed on a remote device, a distributed stream is created. Only devices that have a reference to the sink or source can access the data, or emit data on it. The dedicated stream approach avoids the network overhead of multicasting.

Listing 39 is an example of a dedicated stream. The smartphone creates a local sink (line 29) and references it in the first-class programs (line 10). Each thermometer will emit its data into the sink, hereby creating a dedicated stream between the smartphone and each thermometer. As soon as the smartphone disconnects or the sink terminates, the streams on the thermometers will terminate as well. If the smartphone reconnects to the network, the thermometers will be updated again, and a new average will be computed.

4.5 Separating Concerns in Potato with Creek

We have introduced the meta-level architecture of Creek in chapter 3. The raison d'être for Creek and its meta-level architecture, μCreek, is to address the non-functional concerns in CPSs. In chapter 3 we showed that stream Domain Specific Languages (DSLs) suffer from hardcoded propagation semantics, entanglement of functional and non-functional operators, and that the non-functional operators are implemented ad-hoc in stream DSLs.

In chapter 1 we argued that one of the challenges of creating CPSs is that the heterogeneity of devices introduces differing non-functional concerns. These differences make it difficult to integrate the devices into a single Potato application, because the application code has to be adapted to cater to these differences. With "Separation of Concerns" (SOC) in mind, it is essential that these non-functional concerns are addressed separately and do not affect the application logic. In this section, we show that the **meta-level architecture of Creek** can be used to **express non-functional concerns** in the context of a CPS. Non-functional differences are extracted from the that are set up between them in the Potato application. The most prominent non-functional concerns in CPS are security [71], latency and cost [7], Quality of Service (QoS) [9], and interoperability and protocols [81].

In what follows we show that the non-functional concerns of devices can be evacuated into a meta-stream using Creek. For example, security can be addressed by encrypting the communication between two entities. Communication costs can be addressed by batching or compressing the data sent between entities or changing the propagation protocol to pull-based semantics. Interoperability of different data types can be increased by implementing serializers at the meta-level, facilitating communication between entities that use other protocols.

Case Study: Multiple Encodings The following is an example of a CPS with non-functional concerns. The VUB smart campus is equipped with thermometers to control the central heating system. The HVAC systems require a stream of average temperatures on campus to control the heating and cooling. There are two types of thermometers, each with their own non-functional concerns. Type A thermometers emit their measurements encoded in JSON, and type B thermometers emit their measurements encoded in XML.

To obtain an average of these thermometers, the data must first be decoded into a floating point value. The obvious solution is to insert a `map` operator after the thermometers that encodes the data depending on the type of the thermometer. However, this would mean adapting the application level code, making the application less adaptable. **If a new type of thermometer is added it would require changes to the application.** The application should be written as a transformation of floating point numbers. An update could be deployed on the thermometers that provides a stream of floating points, however, this update is considered part of the application code, resulting in entangling of concerns. The different encoding schemes can be addressed at the meta-level of the stream between the thermometer and the device that computes the average. Each value on the stream can be inspected for its encoding. Depending on the encoding, the value is decoded at the meta-level, and the floating point is propagated to the base-level. This allows the entire stream to assume the values to be floating points, regardless of their original encoding. Figure 4.5 show the graphical representation of these two streams. Type A emits JSON encoded values, and type B emits XML encoded values. The `average` operator requires floating points, however. Listing 40 shows the partial DAG of the meta-level that decodes the values. For each value that is sent from a source, it will be decoded appropriately. If the value is not encoded, it is emitted without decoding. The rest of the meta-level leaves the stream unmodified. The full implementation can be found in appendix F.

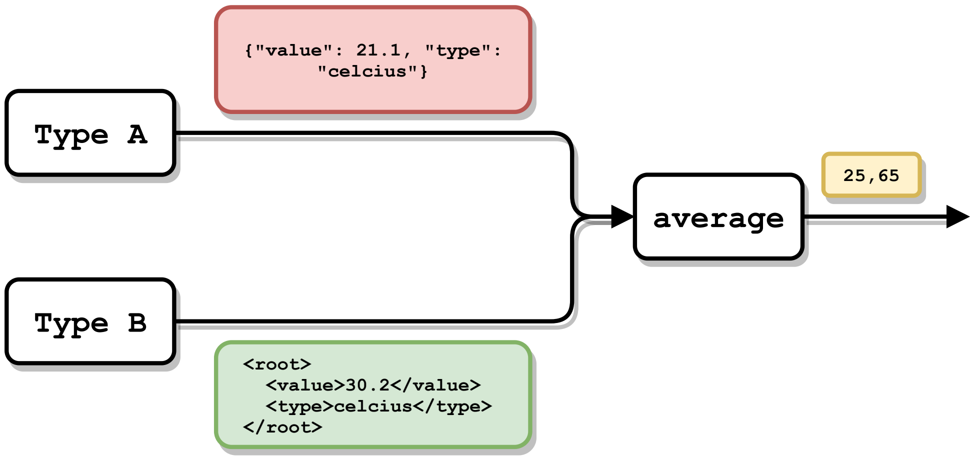

{"value": 21.1, "type": "celcius"}
Type A
average
25,65
Type B
<root> <value>30.2</value> <type>celcius</type> </root>

Figure 4.5: *Graphical depiction of the stream of two thermometers and their average.*

In conclusion, the meta-level of the stream DSL can express the non-functional concerns mentioned in the state of the art regarding communication between devices. They separate the application level from the non-functional concerns and ensure that an application is only concerned with the application logic. Without the meta-level framework from Creek, the application for the thermometers would have to be adapted to suit the integration of multiple types of thermometers.

```
dag next as filter(&match?({_, :next, _, _}, &1))
                ~> map(fn {state, :next, encoded, from} ->
                  decoded =
                    cond do
                      is_xml?(encoded) -> xml_to_float(encoded)
                      is_json?(encoded) -> json_to_float(encoded)
                      true -> encoded
                    end

                  {state, :next, decoded, from}
                end)
                ~> base()
                ~> effects()
```

Listing 40: *Decoding the values coming from a source.*

4.6 Related Work

Potato is a middleware to design Cyber-Physical System (CPS) applications for large-scale distributed systems with heterogeneous devices. CPS is a broad term that can mean different things, depending on who defines the term. In what follows we situate Potato in the state of the art in CPS, and the related fields of Internet of Things (IoT) and Wireless Sensor Network (WSN).

4.6.1 Wireless Sensor Networks

Wireless Sensor Networks (WSNs) are large-scale distributed systems with the aim of gathering data as efficiently as possible from a physical environment. WSNs are considered to be a precursor the Internet of Things (IoT) and Cyber-Physical System (CPS). The challenges WSNs address overlap with the challenges present in CPS, such as code mobility, large scale, and network failures. We discuss the most prominent approaches to overlapping challenges here.

Sensor nodes ("motes") – small battery powered microcontrollers equipped with a few kilobyte of memory, a sensor, and radio communications – are scattered over an area and continuously sense their environment. A central base station aggregates the node's measurements for offline analysis. *Sense-only* systems [74] are built to read their environment and send the data to the central base station. The reference topology for a sense-only WSN consists of a handful to hundreds of wirelessly connected homogenous sensor nodes. Examples of sense-only WSN systems are avalanche prediction [38], animal tracking [44], and flood detection [11]. The networks are mostly static in topology, but applications exist that contain mobile nodes, e.g., trackers on animals. Wireless Sensor and

Actuator Networks (WSANs) create a control loop where the base station gives commands to the sensors to act on their environment [8]. WSANs blur the line between WSNs, CPSs, and IoT. CPSs are systems that revolve around the exchange of data and instructions.

All nodes in a WSN application must operate as long as possible with as little resources as possible. To achieve this, the software on the nodes is optimized to optimize the lifespan of the nodes (e.g., deep sleeping and energy harvesting). The downside of this approach is that the application logic is often *implementation-driven*, and requires expert programmers. One of the earliest operating systems for nodes are TinyOS [65] and Contiki [37], and applications are written in languages such as C, or NesC [45]. These applications operate close to the metal, and have to deal with concerns such as radio locking for data transmission and manual memory management. Potato is an *application-driven* approach; the framework is designed to cater to the programmer, rather than the hardware.

Several research efforts raise the level of abstraction to program WSNs. Regiment [76] is a high-level programming that works on reactive signals. A signal is a designation of sensors in the network that is created using geographic constraints, e.g., “all nodes within a 1 km radius.” The value of the sensors is sampled manually at the base station, creating a stream of data. The work by Madden et al. [67] represents the whole network of sensors *hollistically* as a database that can queried using SQL-like expressions. A compiler optimizes the queries into the most efficient code to obtain the data from the sensors. TeenyLime [28] is an event-driven middleware that introduces asynchronous communication via tuple spaces. Applications consist of callbacks that execute based on specific events. Sensors emit data on the network by placing tuples in the distributed tuple spaces. The event-driven approaches in WSN are either callback-based approaches, or precompiled approaches. The high-level approaches offered here are often at a higher abstraction level than Potato, but are also limited to read-only approaches. In Potato the data streams can start at any device and end in any device in the network, even when these devices are not known at compile-time.

Updating motes at runtime in a WSN is useful for two reasons. First, it allows the application to adapt to changing requirements and allows nodes to localize computations to reduce network traffic. All approaches achieve this with a form of mobile code. Agilla [40] introduces mobile *agents* to move computations from the base station to the nodes. Agents are pieces of application logic written in assembly. An agent can be defined with weak mobility or strong mobility [43]. ActorNet [64] is a higher level actor framework that uses Scheme-like syntax to define actors. In ActorNet data is not sent across the network. An agent designates the nodes it requires for its computation, and travels to all these nodes to evaluate the computation locally. ActorNet actors feature a managed runtime and application-level multitasking. The code mobility in Potato is a means to setup distributed data streams between devices, rather than to contain application logic. The mobile actors approach still scatters application code into different mobile actors, that execute application logic on the device the actor resides in. Using Potato, code mobility and distributed data streams offers more expressivity in defining where computations are executed, as well as which devices are involved.

Wireless Sensor networks are optimized to work with stringent resources, massive scale, and highly volatile networks. The hardware of the sensor nodes ("motes") is mostly designed in function of the application. WSNs are similar to CPSs in that they are large-scale distributed systems. Both systems consist of tens or hundreds of nodes. Applications on these networks are maintained remotely, because physical access to the sensors is often impossible and time-consuming. Both WSNs and CPSs are designed with failure as a given, however, a WSN is not impacted as much by a failed sensor due to its large-scale and homogenous nature, where a CPS must *heal* to provide the necessary services. CPSs consist of heterogeneous devices that are designed to integrate into a wide variety of applications. WSNs consist of homogenous devices that all deploy the same software.

4.6.2 Internet of Things

The IoT is an umbrella term for frameworks that consist of various physical devices that are connected to the internet to exchange data, and expose services to their users. The first IoT applications were based on RFID tags [1]. Since then, IoT has evolved into various types of systems, ranging from smart home applications to ultra-large-scale applications consisting of thousands of devices [81]. The common denominator of IoT applications is a user who interacts with devices, and devices that interact with each other or the internet. CPSs can be considered a specific type of IoT applications that focusses on machine-to-machine communication, are not always connected to the internet, and have an integration into the physical world.

Most research in **the IoT space focusses on systems research** such as networking systems, hardware, and network protocols [18]. We limit ourselves to approaches that focus on programming languages, or frameworks to design applications. The main challenges in IoT applications are resource discovery, resource management, data management, code management, large scale, dynamic

[1] `https://www.rfidjournal.com/that-internet-of-things-thing`

networks, spontaneous interactions, and distributed computations [81]. In summary, IoT applications are very similar to CPSs, but they differ in the types of applications that are deployed on the network. CPS applications are more autonomous and data-driven than IoT applications, which typically involve human interaction.

To deal with the dynamic and open network topology, service discovery is a crucial element in any approach. Approaches such as ElIoT [87] **tag each device with a label**, e.g., `printer`. A device can multicast messages to a tag, without knowing the devices. This primitive allows creating designation based on the tags. Each device in an ElIoT application is tagged with a type, e.g., `printer`. AmbientTalk [34] has the same approach, but avoids explicit discovery with callbacks when a specific type of device is discovered on the network. Potato draws inspiration from this approach but avoids the callback-based approach by using stream-based programming. DDS [78] and Rx4DDS [58] offer automatic **service discovery, but rely on a *seed list***. When devices join the network, a central server provides an initial list of devices to bootstrap service discovery. Cuttlefish [77] uses a **fully centralized approach**. A *manager* device serves as a relay for communication between all devices, and keeps a list of devices in the network. Potato uses distributed service discovery, avoiding a single point of failure and preexisting infrastructure, but burdens devices with network discovery.

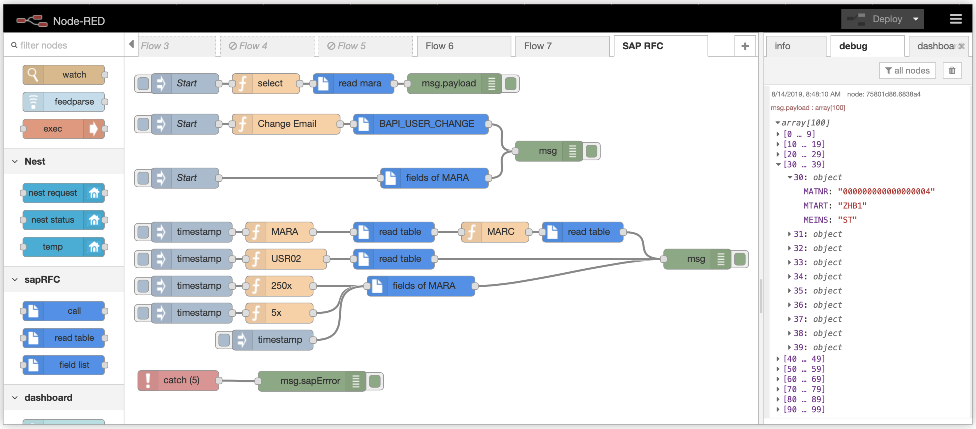
The distributed computations in an IoT system require devices to identify and communicate with other devices in the network. A device needs to *designate* the devices it needs to provide a given service. MundoCore [6], Calvin [80], Mobile Gaia [27], and Srijan [79] take a ***hollistic* approach.** The whole network is treated as an abstract machine, and the runtime is tasked with ensuring the right services are deployed on the right hardware. A service defines which other services it relies on, and the runtime ensures that the connections between them are made when they are deployed. To allow the system to deploy the right services on the right hardware, programmers have to provide information regarding the devices, their capabilities, and the requirements for the services. Achlys [62] **decouples devices in space and time** with a *task CRDT*. Devices ask for data by inserting a *task* into a distributed CRDT. Devices who can fulfill a task replace it with a result in the CRDT. AmbientTalk [34] implements *ambient references* that allow a device to designate based on tags, and refine that designation with runtime properties. The high-level approaches such as MundoCore define dependencies between types of devices at compile-time, and the **runtime automatically sets up the data exchange between the devices**. This approach is rigid in the sense that **all types of devices must be known at development time**. Achlys its CRDTs are highly resilient in their data exchange, but are focussed on the exchange of computations, rather than the setup of data streams. Potato's code mobility and stream-based programming is geared towards distributed data streams.

The variety of applications, and the open nature of the network make it hard for any device to provide all possible services from the start. Therefore, **dynamic provisioning of devices is necessary to create an extensible network**. SOCRADES [49] has a *Service Repository* in the network that offers application logic to devices. When a service is required but not available, the runtime will instruct a device with the right capabilities to **install a service from the service repository**. Calvin [80], GREEN [86], and ElIoT [87] allow the **transmission of compiled binaries across the network**. Calvin transmits compiled jar files, GREEN transmits Java class files, and ElIoT transmits compiled Erlang files. This is a form of *weak code mobility*, because there is **no execution context or scope that is transmitted with the application**. At the other end of the spectrum there is *full mobility*, where **running computations can be migrated by the runtime**. Most of these approaches fall under the mobile agents category. MundoCore [6] allows a deployed service to migrate to another device, if it has the correct capabilities. The runtime does this without any intervention of the user or interrupting the computation.

In Potato the application is defined as first-class reactive programs that are deployed at runtime. The programmer is tasked with explicitly transmitting programs to other devices, which is less elegant than on-demand service deployment.

The non-functional concerns in IoT applications are present in many layers, and the approaches vary. GREEN [86] is a component-based middleware with a reflective layer to manage the deployment of components at runtime. This layer separates the deployment logic of components from the application logic. AmbientTalk [34] is a language for mobile ad-hoc networks, and has a sophisticated meta-level architecture. In AmbientTalk the semantics of an object, actors, and method calls can be intercepted and modified. This opens up nearly unlimited possibilities to adapt the language to a specific need. While not specifically geared towards the non-functional concerns in context of data streams such as Creek, it can be used to express a wide variety of different semantics in the programming language. Part of the differences in non-functional concerns in these types of applications stems from the heterogeneous devices. Approaches such as Srijan [79], MundoCore [6], and Calvin [80] abstract the devices such that these differences are hidden behind a generic interface. This approach facilitates integration of heterogeneous devices, but lacks extensibility in which concerns it can abstract over.

We proposed reactive programming as the paradigm to express distributed event-driven systems. In the state of the art, few approaches already employ reactive programming. Node-RED [84] is a visual programming language that is reactive, or *flow-based*. An application can have one or more data sources, and one or more data sinks. Each transformation to the incoming data is represented as box, and edges between the boxes define the data flow. Figure 4.6 is a screenshot of a Node-RED application. The streams in a Node-RED application are limited to local execution. Distributing a stream must be done by explicitly connecting to other devices via TCP or MQTT. Node-RED is highly popular in the hobby community due to its low barrier to entry. NR-D [46] is a fork of Node-RED that implements distributed streams. Edges can be drawn between boxes that are deployed on different machines on the network. AmbientTalk/R [24] is a fork of AmbientTalk that adds reactive *signals* to the language. When a reactive signal changes value, it updates all the computations that depend on it. Reactive signals can be exported across the network to create distributed reactive programs. Rx4DDS [57] is a fork of DDS that adds Reactive Streams [2] to DDS applications. Similar to Node-RED, the streams as limited to local execution, and cannot cross the network boundary. Potato was designed around Reactive Streams from the ground up and uses first-class transparently distributed streams as the mechanism for data dissemination and dynamic updates.



```
8/14/2019, 8:48:10 AM   node: 75801d86.6838a4
msg.payload : array[100]
▾array[100]
 ▸[0 … 9]
 ▸[10 … 19]
 ▸[20 … 29]
 ▾[30 … 39]
   ▾30: object
      MATNR: "000000000000000004"
      MTART: "ZHB1"
      MEINS: "ST"
   ▸31: object
   ▸32: object
   ▸33: object
   ▸34: object
   ▸35: object
   ▸36: object
   ▸37: object
   ▸38: object
   ▸39: object
 ▸[40 … 49]
 ▸[50 … 59]
 ▸[60 … 69]
 ▸[70 … 79]
 ▸[80 … 89]
 ▸[90 … 99]
```

Figure 4.6: *Example Node-RED application.*

4.6.3 Conclusion

Table 4.1 summarizes our literature study. Each row in the table represents an approach, and each column maps onto one of the challenges of CPSs. We summarize the columns below.

- **Designation** indicates what type of designation is used to address devices in the network. We have found approaches that use intensional, extensional, or physical addressing.

	Designation	Service Discovery	Reactive?	Offloading	UPM	NFC
Potato	Intensional	Distributed	✓	Semi-Strong Mob.	Streams	✓
AmbientTalk	Intensional	Distributed	✗	Weak Mob.	Event-Driven	✓
AmbientTalk/R	Intensional	Distributed	✓	Weak Mob.	✗	✓
DDS	Extensional	Seed List	✗	✗	✗	✗
Rx4DDS	Intensional	Seed List	✓	✗	Streams	✗
ElIoT	Extensional	Distributed	✗	✗	✗	✗
Regiment	Extensional	Distributed	✓	RPC	✗	✗
Node-RED	✗	✗	✓	✗	Visual	✗
Calvin	Extensional	✗	✗	Strong Mobility	✗	✗
Achlys	Extensional	✗	✗	Weak Mob.	✗	✗
GREEN	Intensional	Distributed	✗	Weak Mob.	✗	✓
Cuttlefish	Intensional	Centralized	✗	✗	✗	✗
MundoCore	Intensional	Centralized	✗	Strong Mob.	✗	✗
Srijan	Intensional	Centralized	✗	✗	✗	✓
ActorNet	Intensional	Centralized	✗	✗	✗	✗
Mobile Gaia	Intensional	Centralized	✗	✗	✗	✗

***NFC**: Non-functional Concerns, **UPM**: Unified Programming Model, **RPC**: Remote Procedure Calls*

Table 4.1: *Summary of related work with respect to requirements for CPSs.*

- **Service Discovery** indicates the type of service discovery that is used. An approach can use distributed, centralized, seed list service discovery, or lack service discovery entirely. Distributed service discovery requires each device to monitor the network. Centralized discovery has a centralized service that is the source of truth for devices in the network. Seed list discovery deploys devices with a pre-defined list of centralized services that can be used to bootstrap decentralized discovery.

- **Reactive** indicates whether the approach employs reactive abstractions (e.g., streams) or not.

- **Offloading** indicates if the approach makes it possible for devices to distribute computations across the network. This approach can be code mobility, mobile agents, or remote procedure calls.

- **UPM** indicates if the approach offers a unified programming paradigm or mixes paradigms to combine approaches to address individual challenges of CPS design.

- **NFC** indicates if the approach takes non-functional concerns into account or not.

Conclusion This section discussed the state of the art regarding designation, service discovery, reactive frameworks, code mobility, unified programming models, and non-functional concerns that apply to CPSs. We observed that designation and service discovery are well-studied, and most approaches offer a form of service discovery that abstracts over communication-oriented methods.

The approaches that lack service discovery typically prohibit dynamic networks, too. The technical aspect of service discovery is not or should not be an issue in modern software engineering. However, integration into the programming language or framework is essential. Approaches such as AmbientTalk and GREEN integrate the service discovery properly, while approaches such as ElIoT require the application to manually query devices without any support from the middleware. Reactive programming has become popular in the last decade. In the context of CPSs, not much has been done yet, while reactive programming is designed for event-driven systems. Code mobility is a concept that has been well studied in the past, however, its applications for reconfigurable distributed systems and real-time updates have mostly been studied in context of Mobile Agents. A unified programming model was not found in any middleware, except Rx4DDS, although limited to local applications only. Finally, the addressing of non-functional concerns through meta-level programming has not been found in any middleware or frameworks for CPSs. AmbientTalk is the only approach that incorporates a meta-level architecture, and it is designed according to the design principles set forth by Bracha and Ungar [20].

4.7 Evaluation

In this section we discuss our evaluation of Potato by implementing the "VUB 4.0" scenario introduced in section 4.1. The use case is a more elaborate adaption of a recurring Smart Office example often found in literature[26, 87, 12]. In order to provide more meaningful qualitative analysis, we implement the same application in a language (Elixir) without the abstractions offered by Potato and Creek. To measure the expressiveness of Potato over Elixir, we categorize the resulting code into six categories. We conclude by discussing the patterns that appear to address the challenges in Cyber-Physical System (CPS) in the plain Elixir implementation, and how they compare to Potato.

4.7.1 Potato Compared To General Purpose Language

We compare our Potato implementation of the three scenarios from section 4.1 with an implementation in plain Elixir. We taxonomized the resulting code into the following six categories by classifying each individual line of code. Each line of code is given a unique color, as shown in fig. 4.7.

- **Application Logic** is code that implements the actual behavior of the application. This code is specific to the application it's functional requirements and typically cannot be reused in other applications.
- **Data Routing** is code that is concerned with setting up data between different devices in the network. This includes the code to manage the

recipients of data and the senders of data.

- **Data Flows** is code that is concerned with transformation and dissemination of data in the network. This includes the logic requires to continuously measure data and the transformations on data.
- **Addressing & Discovery** is code that is concerned with the discovery and designation of other devices in the network, and identification of devices.
- **Non-Functional Code** is code that is concerned with the non-functional requirements of the data streams in the application.

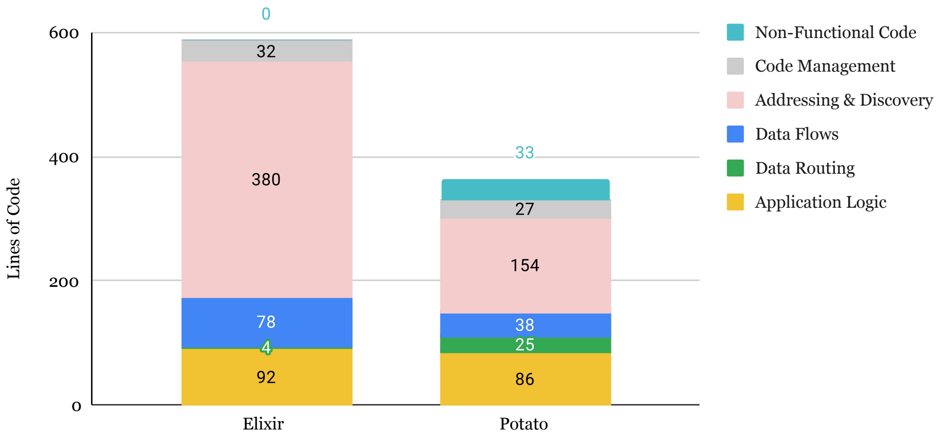


Figure 4.7: *LoC per category in regular Elixir and in* Potato*.*

Figure 4.7 depicts the categorization of each line of code from an implementation in plain Elixir and an implementation using Potato. Ideally, we would expect the bar to only contain application logic, but that is not practically possible. The columns in fig. 4.7 show the total lines of code per category. The highlighted code can be found in appendix appendix H.

Service Discovery The most noticeable difference is the code dedicated to service discovery and designation. The Elixir application requires more than double the code to discover and designate devices on the network. In Potato, the service discovery happens in the background and is conceived as a stream of events. In regular Elixir, this behavior has to be mimicked by exchanging messages and takes up much additional code.

Data Flows The data flow code is concerned with generating and manipulating data streams in the application. Without the use of Potato, each data source has to be defined as an Elixir process that loops indefinitely to generate a stream of data. Each process must manually send its data to its “subscribers”. To do so, a process must keep track of its subscriptions, and must allow processes to subscribe at runtime. Finally, each process must explicitly express its interest in another `data stream` by sending it a message. These requirements make the manual data streams a brittle construction requiring extra code. Moreover, in Potato, distributed data streams are monitored and managed by the Stream Termination Protocol (see section 2.3). These guarantees are absent from the plain Elixir application. The behavior could be mimicked with monitors and links (see appendix A) in Elixir, but that would be a reimplementation of Creek; one that pollutes the application logic.

Data Routing Data routing is the only category that requires more code in the Potato application. The data routing code concerns setting up and distributing data sources and sinks across the network. In plain Elixir, it is easier to set up ad-hoc distributed processes. In Potato, however, a stream must be deployed with a source and a sink, even if the data is not processed any further by the sink. This manifests itself through `ignore` sinks throughout the application. However, the distributed data flow setup by Potato is properly terminated in the case of a network partition. In contrast, in the case of the plain Elixir application, there would be processes running on the devices that would never be terminated. Code to address this issue is not present in the plain Elixir implementation, and would make the implementation even more complex.

Non-Functional Concerns The non-functional code in the VUB application is limited to scenario 2, where the sound level meters emit data encoded in both JSON and XML. In the Elixir application, this code is part of the application, so the effective lines of non-functional code are 0. In the Potato application, however, there are 33 lines of code to handle the non-functional concerns.

Conclusion We conclude that Potato introduces a logical overhead when setting up data routing sources and sinks, depending on the application. However, the Potato data streams are managed by Creek to ensure that they are cleaned up in case of network failures. Service discovery, code management, and data flows are expressed in much less code than in plain Elixir.

4.7.2 Recurring Patterns

When comparing the codebase of both implementations presented in section 4.7.1, there are a few recurring patterns that pop up when addressing service discovery, code management, data flows, and data routing. We discuss these patterns to show how exactly the abstractions offered by Potato have to be reconstructed in a general-purpose programming language that does not have the necessary abstractions for CPS applications.

Service Discovery Service discovery is the logic that is concerned with monitoring the network and executing application logic in case a specific device is discovered. In section 4.3.2 we showed how the `world stream` in Potato directly exposes the network events at the application level to cater to this requirement. In the Elixir implementation, the programmer has to implement three parts. First, a monitor process that monitors the network and connects via TCP/IP to other Elixir nodes on the network. Second, a process to handle events generated by the monitor service to filters out relevant events. And finally, an action to execute when a relevant device connects to the network.

Listing 41 shows the skeleton code that is required for each discovery request made in the application logic. For every designation, a process must be spawned that explicitly subscribes to events from the discovery process. Next, a predicate and action are defined to filter out required devices and the action that must be taken once they are discovered. The `discover_loop` loop handles requests the description (*node descriptor* in Potato) of the remote device. The predicate determines if the device is of interest and then executes the given action, and then the loop repeats. The equivalent code in Potato is shown in listing 42. In Potato, the programmer uses a `filter` operator to filter out the necessary devices, and a `map` operator to apply the necessary logic to the device.

Without Potato

```
def discover_loop(pred, action) do
  receive do
    {:discovered, device} ->
      get_remote_description(device)

      receive do
        {:description, description} ->
          if pred.(description) do
            action.(device)
          end
      end
  end

  discover_loop(pred, action)
end

spawn(fn ->
  # Register for updates on discovered devices.
  send(Process.whereis(Discovery), {:on_discover, self()})

  # What should  the device have as properties?
  predicate = fn device_properties ->
  # ...
  end

  action = fn remote ->
  # ...
  end

  discover_loop(predicate, action)
end)
```

Listing 41: *Skeleton code to discover devices on the network in Elixir.*

Potato

```
world_stream
~> filter(fn ->
  #..
 end)
~> map(fn device ->
  #..
 end)
```

Listing 42: *Skeleton code to discover devices on the network in* Potato*.*

The service discovery in plain Elixir offers the same functionality as the `world stream` and Creek in Potato. However, the pattern in listing 41 is not extensible to quantify the desired amount of devices to be discovered. For example, in the second scenario of the “VUB 4.0” application, the professor’s laptop only discovers the beamer once to turn them off, to allow the professor to turn on the beamer afterwards. The discovery is limited to a single instance of the device, and then stops. Otherwise, the beamer would be turned off every time the professor tries to turn it on.

Data Routing In Potato, the routing of data is done implicitly. When a Directed Acyclic Graph (DAG) is deployed, its sources send data through the stream and end up in one or more sinks. In Elixir, this data routing is explicit by sending ad-hoc “subscribe” messages to other processes. Any process that wishes to emit data has to broadcast data to these subscribers explicitly. Listing 43 shows the skeleton code of a process that can be subscribed to. The equivalent code in Potato is shown in listing 44.

Compared to Creek, the logic in listing 43 lacks important bookkeeping logic such as unsubscribing, process termination when no subscribers are left, and failure handling. Adding all these features would result in a reimplementation of Creek in Elixir at the application level.

Without Potato

```
F.loop_async(
  fn {state, clients} ->
    receive do
      {:subscribe, pid} ->
        {state, Enum.uniq([pid | clients])}

      {:value, value} ->
        #...
        for client <- clients do
          send(client, message)
        end
    end
  end,
  {state, []})
```

Listing 43: *Process that can be subscribed to in Elixir.*

Potato

```
Creek.Source.function(fn -> ... end)
```

Listing 44: *Process that can be subscribed to in Potato.*

Failure Handling In section 2.3 we explained that the stream termination protocol ensures that distributed streams are cleaned up when a network partition occurs. The stream termination protocol implementation relies on the link and monitor mechanisms present in Elixir (see appendix A). In the plain implementation of the VUB application these monitors have to be manually created and monitored by each process.

Listing 45 shows the code needed to monitor another process. First a monitor has to be created, and secondly, the process has to monitor its message inbox for messages that signal the monitored process is offline. This logic is entangled with the application logic. In Potato none of this logic is necessary, and the stream termination protocol handles all the monitoring, as is shown by the equivalent Potato code in listing 46. All operators in Potato, by default, monitor their up- and downstreams for failure. When a failure occurs the runtime takes care of removing the connections, and terminating the stream if necessary (see section 2.3).

```
F.loop_async(                                                   Without Potato
  fn {state, clients} ->
    receive do
      {:subscribe, pid} ->
        Process.monitor(pid)
        {state, Enum.uniq([pid | clients])}

      {:DOWN, ref, :process, pid} ->
        {state, Enum.reject(clients, &(&1 == pid))}

      {:value, value} ->
        #...
        for client <- clients do
          send(client, message)
        end
    end
  end,
  {state, []})
```

Listing 45: *Linking to subscribed processes.*

```
world_stream                                                    Potato
~> filter(fn ->
#..
end)
~> map(fn device ->
#..
end)
```

Listing 46: *Skeleton code to handle failures in a stream in Potato.*

Mobile Code Deployment Mobile code deployment is at the heart of Potato. It is the only mechanism to execute code on remote devices, and to distribute streams across the network. Elixir offers semi-strong code mobility, which makes the process of remote code deployment as simple as it is in Potato. Listing 47 show the code necessary to deploy a function on a remote node in Elixir, and listing 48 shows the equivalent code in Potato. The assumptions here are that the remote device is connected to the network.

The ad-hoc deployment of processes on remote nodes makes it impossible for the runtime to manage them. In Potato, each program is deployed by sending on the `update stream` of a remote device. The receiving device has full control over the deployment process.

Without Potato

```
Node.spawn(remote, fn ->
          #...
        end)
```

Listing 47: *Deploying a function on a remote device in Elixir.*

Potato

```
p = program do
      #...
    end
next(remote.update_stream, p)
```

Listing 48: *Deploying a program on a remote device in Potato.*

4.7.3 Memory Footprint

In what follows we compare the memory overhead of a Potato application and a plain Elixir implementation. In order to compare the memory footprint of Potato to plain Elixir, we inspect the second scenario from section 4.1 (i.e., “Noisy Campus”) in more detail.

The “Noisy Campus” scenario contains a smartphone that maintains connections with multiple sound level meters and windows. This network can be easily scaled in terms of network size by adding more windows or sound level meters. Additionally, the Potato application for the sound level meters also contains meta-level behavior to integrate various data representations, taking the memory impact of the meta-level streams into account.

We first measure the memory footprint of the plain Elixir implementation for different network sizes. Next we measure the Potato implementation for the same network sizes and compare the memory footprint.

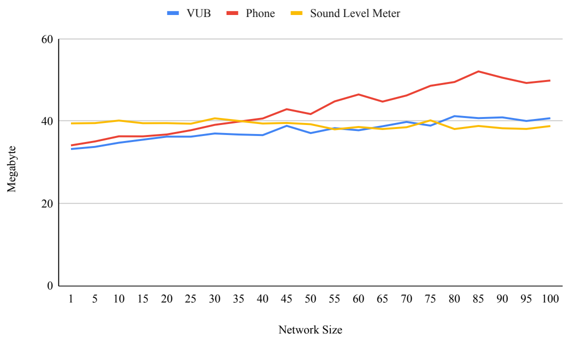


Figure 4.8: *Memory consumption in megabyte for Phone, VUB, and Sound Level Meter devices with varying network size in Potato.*

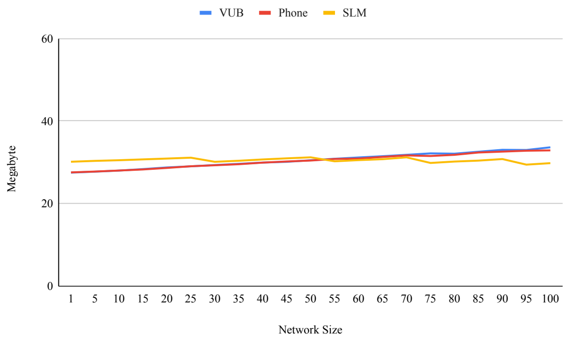


Figure 4.9: *Memory consumption in megabyte for phone, VUB, and sound level meter devices with varying network size in Elixir.*

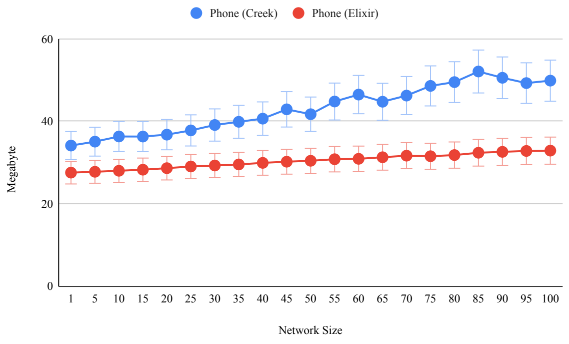


Figure 4.10: *Memory consumption in megabyte for phone devices in Elixir and Potato.*

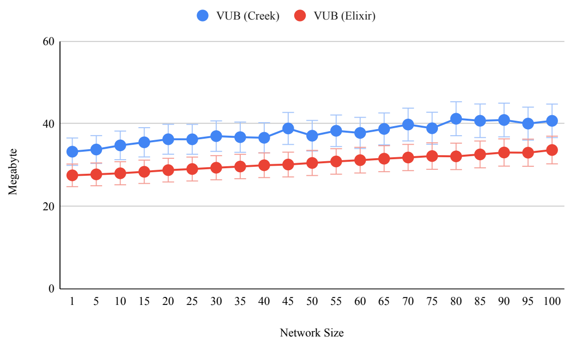


Figure 4.11: *Memory consumption in megabyte for VUB devices in Elixir and Potato.*

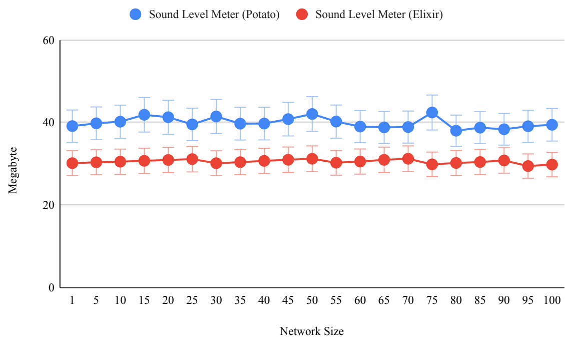


Figure 4.12: *Memory consumption in megabyte for sound level meter devices in Elixir and Potato.*

Figure 4.9 and fig. 4.8 show the total memory footprint of the smartphone, VUB computer, and sound level meters. Looking at the memory footprint of the Elixir implementation in fig. 4.9, we see that both the phone and VUB device consume more memory as the network size grows. The same pattern can be seen in the Potato chart (fig. 4.8). This pattern is explained by the fact that as the network grows, the VUB and smartphone need to maintain additional connections to the devices. The VUB device connects to the device to update it and then disconnects, but the smartphone maintains a constant connection to the device.

The sound level meters do not consume more memory when the network size grows. This can be explained by the fact that regardless of the network size, the sound level meters only stream data to the smartphone. At any network size, sound level meters only have a single outgoing stream. The VUB and phone devices need to communicate with all the sound level meters, thus increasing their memory footprint as the network grows.

Figure 4.10 gives a more detailed view of the memory consumption of the smartphones in the Elixir and Potato implementation. The plain Elixir implementation requires less overhead per device compared to the Creek implementation. This means that a stream between two devices in Potato requires more memory than a stream in Elixir. A stream in Potato consists of multiple Elixir processes that represent individual operators in a stream. Depending on the complexity of the stream, the constant overhead per connection increases. In the Elixir implementation the entire computation is encapsulated in a single actor. This explains the larger overhead per connection in Potato. Figure 4.11 shows a simliar pattern, which is also explained by the fact that Creek streams consist of multiple Elixir actors. An important difference between fig. 4.11 and fig. 4.10 is that the smartphone has a larger constant overhead compared the VUB device streams. Every sound level meter is streaming data which is transformed at the meta-level, thus incurring an additional overhead per operator in the stream. The streams for the VUB device do not have a meta-level stream and are short-lived. In summary, the streams between the smartphone and the sound level meters have a larger constant overhead compared to the VUB device streams.

Figure 4.12 shows that the memory consumption for sound level meters remains nearly constant, regardless of the network size. This is explained by the fact that an individual sound level meter only maintains a single connection to the smartphone, and therefore does not require more memory as the network grows. The additional memory required in Potato can be explained by the stream that consists of multiple actors, opposed to a single actor in case of Elixir.

Even though Potato and μCreek are an exercise in programming language design, we must conclude that there is a small performance overhead when compared to Elixir, a general purpose programming language. We observed that Potato adds a constant overhead per connection to a device. This overhead can increase further, depending on the complexity of the operations applied to that stream. We conclude that additional work needs to be performed to make Potato applicable to real-world scenarios. However, Potato and μCreek are a first step towards a high-level domain-specific language to express large-scale, unreliable, open systems, such as CPSs.

4.8 Summary of Chapter

In this chapter we have evaluated Potato by implementing a representative use case of a Cyber-Physical System (CPS) and comparing it to an implementation in Elixir, which has no abstractions for CPSs. We compared these implementations by classifying the lines of code according to their concerns. This confirms our hypothesis that Potato and Creek provide abstractions that reduce the accidental complexity in CPS applications. Specifically, service discovery, data flows, and data routing are more elegantly expressed in Potato and Creek. The expression of non-functional concerns was completely absent in Elixir, entangling all non-functional code with the application code. In Potato and Creek, this code is separated using the meta-level architecture of Creek. While our implementation in Elixir had less code concerning data routing, we argue that the expression of data routing in Potato offers failure handling that is completely absent from the plain approach, at the cost of a minor increase in lines of code.

A comparison of Creek and Potato to a state-of-the-art middleware for CPS is desirable. However, related work either does not offer an accessible version of the artifact or does not overlap enough in terms of functionality with Creek to warrant a full implementation of the use case.

We identified four recurring patterns when implementing the "VUB 4,0" use case in a general-purpose language. Failure handling of distributed data communication, routing data between processes and devices, and service discovery on the network. All these patterns are created ad-hoc, entangled with the application logic. We explained the cause of the patterns, and compared them to the equivalent Potato code.

Potato and Creek offer abstractions that make it possible to elegantly express data streams and code mobility in dynamic networks of heterogeneous devices while keeping in mind the modern software design principles. Without the abstractions offered by Potato and Creek, the programmer has to resort to ad-hoc solutions resulting in software of lower quality.

Chapter 5

Implementation

The ideas presented in this dissertation have been supported by two artifacts. This chapter details their implementation. Creek is a macro-based implementation of our stream Domain Specific Language (DSL), and Potato a programming library to aid in the development of Cyber-Physical System (CPS) applications.

We have chosen to implement Creek and Potato in Elixir, a concurrent actor language that is based on the Erlang virtual machine (BEAM). Elixir provides features that lower the barrier for developing distributed frameworks and DSLs. In this section, we discuss the Elixir features we used and how they impacted our prototypes.

5.1 Elixir as a Language Laboratory

The technology stack we used is depicted in fig. 5.1. Elixir is the technical foundation on top of which Creek and Potato are built. μCreek is implemented in Creek. Potato applications are built using building blocks from Creek, μCreek, and Potato.

The Elixir programming language is based on the Erlang virtual machine (BEAM), that was initially built for the Erlang programming language, a bit like Scala was built on top of the Java Virtual Machine (JVM). Elixir can be considered to be a modern incarnation of the Erlang programming language. BEAM has features such as transparent distribution and code mobility that make it interesting for distributed and concurrent applications. Additionally, Elixir has a well-designed macro system, similar to that of Common Lisp. Macros

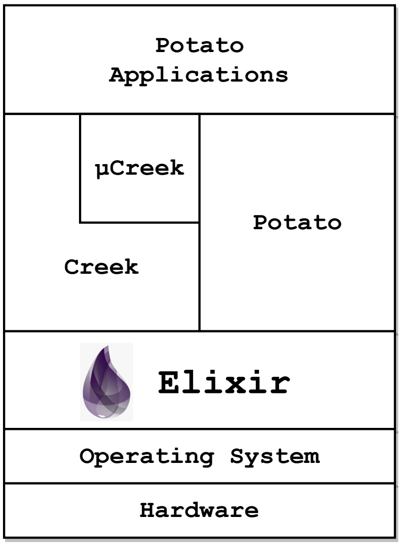


Figure 5.1: *The technology stack used to implement* Creek *and* Potato *in Elixir.*

allow programmers to extend the language to cater to requirements of specific problem domain in ways that would otherwise generate repetitive boilerplate code. As a result, Elixir is a viable choice to design concurrent and distributed Domain Specific Languages (DSLs). In what follows, we briefly introduce the macro system of Elixir, and then discuss the implementation of Creek and Potato in Elixir.

5.1.1 Macros

Macros allow programmers to extend a language with additional primitives. Macros can be used to create an a *embedded domain specific language* (DSL), i.e., a programming language inside a programming language.

As an example of a macro, consider the `unless <test> do: <body>` statement. It's usage is shown in listing 49 (left). Elixir does not have an `unless` statement built-in, and functions cannot be used to implement the desired behavior, because Elixir has *applicative-order evaluation*, so function arguments are evaluated before the body of the function is evaluated. An `unless` statement only evaluates its body if the `<test>` reduces to `false`. The righthand side of listing 49 shows its equivalent `if` expression. Implementing `unless` using functions would mean that the `<body>` of `unless` would be evaluated, even if the test was `true`. Macros, however, are functions that work on the AST at **compile-time**, and can implement the desired behavior.

```
unless 5 != 5 do
    1
end
```

```
if !(5 != 5) do
    1
end
```

Listing 49: *An `unless` expression, and the equivalent `if` expression that the macro (see listing 51) turns it into.*

ASTs Elixir represents its code in the same data structures as its data. An Elixir program its AST is represented as a plain Elixir tuple with three values.

1. The name of the function or expression being called (e.g., `unless`).
2. The context of the expression, which contains line numbers, defining module, etc. We gloss over these details in the remainder of this chapter.
3. A list of AST nodes for each argument

As an example of an AST, consider the code in listing 50 which is the AST of the `unless` expression from listing 49. An `unless` expression takes two operands: the `test` and the body. The function is highlighted in green on line 1, the first operand in yellow on line 2, and the second operand in red on line 3.

```
1 { :unless, [context: Elixir, import: Kernel] ,
2   [
3     {:!=, [context: Elixir, import: Kernel], [5, 5]} ,
4     [do: 1]
5   ]
6 }
```

Listing 50: *The AST of `unless 5 != 5, do: 1`*

Defining Macros Macros in Elixir are defined using two primitives: `quote` and `unquote`. The `quote` function turns its argument expression into an AST, instead of evaluating it. The `unquote` function can be used in an AST expression and takes as argument an AST and injects it into an AST that contains the `unquote` expression.

To show how `quote` and `unquote` work, consider the macro definition of `unless` in listing 51. The macro transforms each `unless` expression into an `if` expression with the `<test>` argument negated. A new AST node to replace the `if` expression is created using the `quote` function. The `test` and body arguments are injected into the new AST using `unquote`.

It is important to note that in this example the macro expansion does not evaluate any arguments. Instead, the transformation happens at the syntax level at compile-time.

```
defmacro unless(test, do: body) do
    quote do
        if(!unquote(test), do: unquote(body))
    end
end
```

Listing 51: *The macro definition of* `unless`*.*

Hygienic Macros The *calling context* is the position in the code where the macro is called. The *macro context* is the place in the code where the macro is defined. Macros in Elixir are *hygienic* [1]. A macro cannot shadow variables that are defined in the calling context. This ensures that macro expansion cannot affect the environment it is called in. The programmer of the macro can explicitly override hygiene if necessary using the `var!` function.

5.1.2 Creek DSL Design

Creek makes use of macros in its Directed Acyclic Graph (DAG) definition language, and its design has been influenced by limitations of Elixir's macro system.

- First, *Elixir macros cannot create custom infix operators*. Function calls in Elixir are denoted in prefix notation, but there are a few infix operators such as + and $\star$. These operators are hardcoded into the tokenizer, and cannot be customized. The Elixir compiler supports a limited amount of predefined and unused infix operators to partially address this issue [2]. In Creek, we have chosen to use ~> (i.e., horizontal composition) and ||| (i.e., vertical composition) as the two primitives to compose DAGs.

- The `defdag` and `dag` primitives in Creek compile the code into a function, because Elixir only allows functions at the top-level in a module. As a result, composing two separate DAGs requires calling their functions. Syntactically, this does not impact the DSL, but it makes it impossible to compile the DAGs at compile-time, and run the μCreek$_C$ truly at compile-time. To address this problem, every DAG is compiled the first time it

[1] `https://hexdocs.pm/elixir/Macro.html`

[2] `https://github.com/elixir-lang/elixir/blob/main/lib/elixir/src/elixir_parser.yrl#L63-L80` [Accessed 5/2/22]

is deployed, and subsequent deployments use a cached version of the compiled DAG.

5.2 Creek Implementation

The main components of Creek are operators, Directed Acyclic Graphs (DAGs), and streams. We describe their implementation in Elixir, discuss features of the language we relied upon, describe how they influenced our design, and we discuss the limitations they imposed.

5.2.1 Implementing Operators

The basic building block of a stream in Creek is an **operator**. An operator describes what should happen to each value when it arrives. In Creek, the operator behavior must be described by three functions for operands: `next`, `error`, and `complete`, two functions for sources: `tick` and `init`, and three functions for sinks: `next`, `error`, and `complete`. These functions return an *instruction* for the runtime indicating what value to propagate. These functions have to respect the canonical stream protocol (see section 2.2.5). Creek enforces the Canonical Stream Protocol (CSP) with a *behavior*, similar to interfaces in object-oriented languages.

Listing 52 shows the Elixir behavior for operators. The behavior states that an operator must define three functions: `next`, `error` and `complete`. Each function takes at least the following arguments.

1. The first argument is always an `Operator` tuple; a tuple that describes the operator behavior, its arity, module implementation, and other meta-data used by the runtime.
2. The second argument is the *state* of the operator. Operator state is used by stateful operators such as the `reduce` operator.
3. The third argument is the *gate* on which the event arrived.
4. The fourth operator in the `next` and `error` functions holds the value and error value, respectively.

The behavior also defines the allowed return values for the functions (lines 3, 5, and 7). These instructions map directly on the canonical stream protocol, with the exception of the `final` instruction. This is an optimization that allows an operator to complete in response to a `next` event, instead of returning `next` and `complete` in response to the next event.

Listing 53 shows the implementation of the `map` operator in Creek as an example. Notice that the return values of each function respect the behavior implementation. If the programmer makes a mistake against the CSP, Elixir's code analyzer will throw a compile error.

```
defmodule Creek.Operator.Behaviour do
  @spec next(Operator.t(), any, any, any)
        :: {any, :next, any} | {any, :skip} | {any, :final, any}
  @spec error(Operator.t(), any, any, any)
        :: {any, :error, any} | {any, :complete}
  @spec complete(Operator.t(), any, any)
        :: {any, :complete} | {any, :error, any}
end
```

Listing 52: *Operator behavior in Creek.*

```
defmodule Creek.Operator.Map do
  def next(this, state, _from, value) do
    new_value = this.arg.(value)
    {state, :next, new_value}
  end

  def error(this, state, _from, err) do
     {state, :error, err}
  end

  def complete(_this, state) do
    {state, :complete}
  end
end
```

Listing 53: *Implementation of the map operator.*

Operator Dictionary Representation An operator expression, e.g., `map(f)`, returns an *operator representation*. An operator representation is a dictionary that contains all the necessary meta-data for the runtime to deploy it in a DAG. Section 5.2.1 shows an example of an operator representation for the expression `map(f)`. The meta-data serves two purposes. First, the arity (`in` and `out`) allows the Creek compiler to verify that a DAG is valid. Second, it contains the arguments that were passed by the user.

If one wants to extend the list of Creek with new built-in operators, a module implementation similar to listing 53 must be given, as well as a description similar to the one below.

```
%Creek.Operator{opts: opts,  type: :operator, arg: f, name: "map",
                ref: Creek.Server.gen_sym(), in: 1, out: 1,
                impl: Creek.Operator.Map}
```

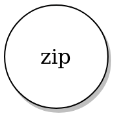


Figure 5.2: *Graphical representation of a map operator.*

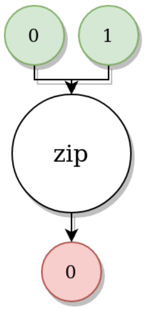


Figure 5.3: *Graphical representation of a map operator as a `GatedDAG`.*

5.2.2 Implementing DAGs

A DAG in Creek is created by composing operators and DAGs using the vertical and horizontal composition operators (i.e., ~> and |||). We have shown how an operator is represented as a dictionary. Next, we discuss how they are composed into a DAG in Creek.

A DAG in Creek is internally represented as a *gated dag* [33]. In a gated dag, each operator is represented by a vertex, and an additional vertex per in- and output gate of the operator. Figure 5.2 and fig. 5.3 show this graphically for the `zip` operator. The `zip` operator has an input arity of 2 and an output arity of 1. Each input is represented as a green vertex with the *port number* in it. Each output is represented by a red vertex with the port number in it as well. Representing an operator in this way allows the Creek compiler to detect if two graphs can be merged, and if a DAG is well defined. At runtime the gated DAG makes it easier to identify from which upstream a datum came, because the ports uniquely identify the source of a value.

5.2.3 Implementing Streams

A stream is a collection of processes in which each represent an individual operator. A stream is created by deploying a DAG. Listing 54 shows an example of deploying a DAG called `factorial`, with a given source actor `src` and a given sink actor `snk`.

```
deploy(factorial, source: src, sink: snk)
```

Listing 54: *Example of a `deploy` statement.*

Deploying a DAG involves the following steps.

1. For each operator vertex in the gated DAG, an *operator process* is created. An operator process is a plain Elixir actor that understands a specific set of messages (i.e., the CSP).
2. The Elixir actors that are provided as sources and sinks are replaced with the actor sockets in the gated DAG.
3. If the DAG was defined with a meta-behavior, the meta-DAG is sent to each actor process.
4. Each process is *linked* to its direct up- and downstream processes. When one of two linked process terminates, a message is sent to notify the other process.
5. The processes representing operators, sinks, and sources, are sent an `initialization` message. After the initialization, sources start emitting data, and operators and sinks can initialize their state. The initialization message acts as a barrier, ensuring no data is emitted while the stream is being deployed.

Operator Process An operator process is a plain Elixir actor that understands a specific set of messages. Listing 55 shows the skeleton code of an operator process. An operator process understands all the messages present in the meta-stream protocol, plus four implementation-specific messages, i.e., `offer_meta`, `add_downstream`, `add_upstream`, and `finish`. Programmers can extend Creek with custom sources and sinks by implementing a custom Elixir process.

5.2.4 Implementing Distributed Streams

Streams in Creek are transparently distributed. A deployment (see section 2.2.4) takes sources and sinks as its argument, and these can be located anywhere on the network. To make this possible, Creek relies on the distribution mechanisms of BEAM. In Elixir, a process has a unique identifier, called its *pid*. This is a primitive Elixir value that can be sent across the network. The transparent distribution of a process in Elixir means that *links* and *monitors* on which the stream termination protocol relies, work regardless of the location a process. In

```
def process_loop(node, upstreams, downstreams, state, meta_state) do
  receive do
    {:offer_meta, meta} ->
        # ..
    {:add_downstream, downstream} ->
        # ..
    {:add_upstream, upstream} ->
        # ..
    {:finish} ->
        # ..
    {:meta_message, m, from} ->
        # ..
    {:initialize} ->
        # ..
    {:next, value, from = {from_pid, _from_gate, gate}} ->
        # ..
    {:complete, from = {from_pid, from_gate, to_gate}} ->
        # ..
    m ->
      warn("Message not understood: #{inspect(m)}")
      process_loop(node, upstreams, downstreams, state, meta_state)
  end
end
```

Listing 55: *The skeleton of an operator actor in* Creek.

other words, no additional engineering was necessary to facilitate data exchange between distributed streams.

However, distribution in Elixir requires it to be part of a *cluster* of computing nodes. To connect to a cluster, a node needs an IP address and port of another machine that is already in a cluster. An explicit connection has to be made to join the cluster. For example, `Node.connect(:"cdt@example.com")` connects to a node running on the server at `example.com`. Setting up a cluster is considered out of the scope of Creek. In section 5.3.2 section we detail how Potato automates cluster discovery.

5.2.5 Implementing μCreek$_R$

The high-level architecture of μCreek$_R$ has been discussed in detail in section 3.4. In this section we discuss the technical implementation of Creek in Elixir.

Defining meta-DAGs A meta-DAG in μCreek$_R$ is defined as a triple of DAGs, a DAG for operators, sinks, and sources. These three DAGs must be defined in a single module with the names `source`, `sink`, and `operator`. Each of these DAGs requires a single source and a single sink. There are no other restrictions on the topology of the DAG.

Loading meta-DAGs When an operator process is spawned, and a meta-DAG is defined, the operator process first creates a meta-source and meta-sink, and deploys an instance of the meta-DAG. The source is a `subject` that the operator process will use to inject meta-messages into the meta-stream. The sink is a `tap` sink, which sends the response of the meta-DAG back to the operator process.

```
def process(node, upstreams, downstreams) do
  if node.meta_dag != nil do
    src = Creek.Source.subject()
    snk = Creek.Sink.tap(self())
    deploy(node.meta_dag, [src: src, snk: snk])
    loop(%{node | meta: src}, upstreams, downstreams, node.arg, nil)
  else
    loop(node, upstreams, downstreams, node.arg, nil)
  end
end
```

Listing 56: *Code to deploy the meta-DAG when an operator process is started. Code is redacted for readability, for full implementation see* `https://github.com/softwarelanguageslab/creek`.

Meta-Messages For every incoming internal message (see listing 55), the operator process creates a meta-message and emits into the meta-source. The operator process blocks until it receives a meta-response from the meta-sink. If a message is processed via the meta-DAG, the operator process does not take any action, other than updating the state of the operator. As an example, consider listing 57, which contain the code that handles a `complete` message in an operator process. The structure is the same for all other internal messages. If no meta-source is present, the process calls the base level (*base stage*) and sends the necessary messages to the downstream operators (*effects stage*) based on the response from the base level. If a meta-stream is present, a meta-event is created, and sent on the meta-source, and the process waits until the meta-DAG emits the new operator state.

```
if node.meta_source != nil do
    meta_event = {%{meta_state: .., ..}, ..}
    next(node.meta_source, meta_event)

    receive do
        {p, :ok} ->
            process_loop(p.node, p.us, p.ds, p.state, p.meta_state)
    end
else
    response = node.base.complete(node, state)
    case response do
        {state, :complete} ->
            send_self({:finish})
            effects_complete(from, downstreams, upstreams, self())
            process_loop(node, upstreams, downstreams, state, meta_state)
        ..
    end
end
```

Listing 57: *Conditional structure to process internal messages in an operator process. Code is redacted for readability, for full implementation see `https://github.com/softwarelanguageslab/creek`.*

Intra-Meta Communication A meta-DAG can send messages to other operators at the meta-level (see appendix E for an example). However, the actor process only understands the canonical stream protocol. To process intra-meta-messages, each operator process must understand a generic `meta_message` message. Whenever an intra-meta-message is sent to an operator, the runtime wraps it in a `meta_message` event. These types of messages are forwarded directly to the meta-DAG, if present. When no meta-DAG is present, they are ignored.

5.2.6 The μCreek$_C$ Compilation Pipeline

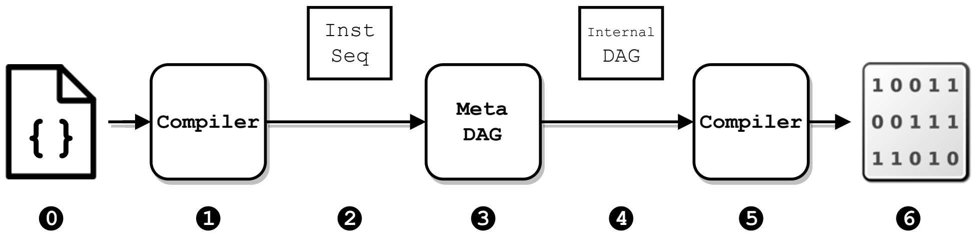


Figure 5.4: *A graphical depiction of the compilation pipeline in μCreek.*

A Creek DAG is defined using one of the macros offered by Creek, i.e., dag or `defdag`. μCreek$_C$ transforms DAGs by streaming them through a meta-DAG in the form of instructions. Both macros execute the following steps for each user DAG at compile-time. The numbers indicate on which stage of the compiler pipeline they map in fig. 5.4.

1. ❶ For each actor socket, a new variable is inserted into the scope where the user DAG is defined (unhygienic).
2. Constant values that are passed to DAGs are inserted into the scope where the DAG is defined (unhygienic).
3. Each actor socket in the user DAG is verified to be either source or sink position.
4. The DAG is verified to be a well-formed DAG (see section 2.2.3).
5. If a meta-DAG is present, the following steps are taken.
   (a) ❷ Topologically sort the user DAG
   (b) For each vertex, create an operator instruction, followed by a name instruction.
   (c) For each edge, create a corresponding instruction.
   (d) ❸ Deploy the compile-time DAG and stream the instructions through.
   (e) Replace the user DAG with the result of the meta-stream.
6. ❹ For each operator in the user DAG, install a copy of the μCreek$_R$ DAG.
7. ❺ Store the compiled user DAG in memory.

Steps 1 to 3 from the above enumeration are executed at compile-time of the Elixir program. Steps 4 to 7 are executed the first time the DAG is deployed, during the runtime of the Elixir program. This is a limitation due to the Creek compiler being written as macros.

5.3 Potato

5.3.1 Implementing Node Discovery

A Potato application makes use of the cluster functionality of BEAM. Distributed nodes can connect to each other, provided that they know a shared secret, IP addresses, and usernames. Setting up a cluster is a tedious task, and Potato automates this process using node discovery. When a device running Potato connects to a shared network, it will automatically discover other Potato instances.

Central to code discovery in Potato is the *discovery process*. When a Potato device connects to a network for the first time, the discovery process announces its presence with a `magic packet`, an UDP datagram [3] that is broadcast onto the entire network. A Potato device continuously listens for magic packets from other devices on the network. When a magic packet is received by another discovery process, both nodes attempt to connect to each other to form a cluster. If the connection is successful, the nodes exchange node descriptors, and the devices their clusters are merged, and form a single Potato application. The application is notified of this change with a `join` network event on the world stream (see section 4.4).

A distributed cluster can become partitioned in case of network failure. Erlang has built-in functionality to monitor nodes in a cluster, similar to links and monitors for processes. The discovery process on each device monitors the cluster to detect network partitions. When a node disconnects from the cluster, the discovery process is notified, and Potato exposes this event to the application by emitting a `leave` event on the world stream.

Preliminary work has been done to extend Potato with *non-IP* networks, such as Bluetooth or NFC. For example, using a Bluetooth low-energy stack, devices can detect other devices in their proximity. We extended Potato to emit two additional types of network events: `near` and `far` events. This is achieved by adding a discovery process that monitors a Bluetooth service, rather than a network service. Bluetooth Potato devices implement an *Eddystone* [4] beacon, and announce their ethernet IP address via bluetooth. When another bluetooth discovery process picks up this signal, it emits a network event that the given device is near. When the devices are no longer in close proximity they both emit a `far` network event.

[3]`https://datatracker.ietf.org/doc/html/rfc919`

[4]`https://github.com/google/eddystone`

5.3.2 Implementing Mobile Code

Potato features first-class reactive programs. Applications can create programs and emit them onto the update stream of other devices to deploy them remotely. Potato exposes an *update stream* to the network that allows remote devices to stream applications to other devices.

The `program` primitive of Potato is macro that captures its body as a delayed computation. The `program` primitive rewrites its body to an anonymous function ("lambda"). There was no further need to modify the code, as Elixir's anonymous functions can be sent across the network and can be classified as *semi-strong mobility* [43]. In brief, code mobility was "free", because the Elixir code mobility features align with the requirements for first-class reactive programs in Potato.

5.4 Summary of Chapter

We developed two prototypes to support the ideas presented in this dissertation. Potato implements the design we proposed in chapter 4, and Creek implements the meta-level architecture we discussed in chapter 2. The implementation of Potato relies on the distributed programming features of Erlang. More specifically, Potato relies on the clustering, node monitoring, transparent distribution of Elixir, and Creek relies on actor links and monitors. Without these features, the implementation would require a significant amount of additional engineering time. Creek and its meta-level architecture μCreek use the macro facilities of Elixir to create the Creek Domain Specific Language (DSL). The macro system of Elixir, in spite of its limitations, makes Elixir into a viable choice as language laboratory. The source code for these artifacts can be found online at `https://github.com/softwarelanguageslab/creek` and `https://github.com/softwarelanguageslab/potato`.

Chapter 6

Conclusion

This chapter concludes this dissertation. We revisit our contributions and how they address the challenges to CPS development introduced in chapter 1. We conclude with the limitations of Potato and Creek and we discuss avenues for further research.

6.1 Revisiting Problem Statement

Cyber Physical Systems (CPSs) are systems that consist of many *heterogeneous devices* and *devices with limited resources*. These *large-scale* systems are *prone to failure*, *open-ended*, and *data-driven*. To some degree, CPSs are similar to Internet of Things (IoT) and Wireless Sensor Networks (WSNs), but focus on autonomous data and instruction exchange. Expressing these data flows in large-scale unreliable distributed networks is difficult with contemporary programming languages. We proposed the stream paradigm as a foundation to address the software design challenges.

Surveying the state of the art, we concluded that the state-of-the-art approaches for CPSs address only a subset of the software design challenges. Some approaches are *implementation-driven*, meaning that the applications are largely influenced by the hardware platform, and often lack high-level abstractions. Implementation-driven applications are mostly written in C or Assembly and have to deal with concerns such as memory management and radio management. *Application-driven* driven approaches abstract away from the lower level with abstractions for communication, discovery, and data routing between devices. However, none of the approaches provide a unified programming paradigm and only tackle a subset of the challenges.

We proposed an application-driven approach that incorporates our approach to all challenges in the stream paradigm. **In our approach the network, devices, programs, non-functional concerns, and data streams are *all* expressed in the stream paradigm.** Concretely, we addressed the following problems.

1. **Large Scale** CPSs can contain hundreds of devices. Writing software that orchestrates data flows between so many devices using traditional communication abstractions is challenging.
2. **Dynamic Networks** Dynamic networks mean that devices can join and leave at unforeseen moments in time. The software has to deal with these changes without interrupting the application.
3. **Heterogeneity** The heterogeneous devices exhibit *non-functional traits* that programmers must address to integrate them into an application. In traditional software engineering, the principle of "Separation of Concerns" (SOC) dictates that different concerns should not be mixed.
4. **Dynamic Networks** The dynamic network means that devices can join and leave at undefined times. The software has to deal with these changes without interrupting the application.
5. **Data-Driven** The devices in a CPS are continuously executing data and instructions with each other. Devices rely on other data from other devices to provide a service.
6. **Limited Computational Power** Some devices in the application are limited in computational power, which limits the computations they can execute, and the services they can provide.

6.2 Summary

In this section we restate our contributions per chapter.

- **Chapter 2** surveyed the state of the art in stream-based Domain Specific Languages (DSLs). Based on the literature, *we proposed a taxonomy for stream DSLs* along four axes: parallelism, phases, propagation semantics, and backpressure. Finally, *the chapter introduces Creek, a prototypical stream DSL* used as a language testbed in Chapter 3. Finally, we define the *communication protocol between operators* in a stream in the context of Creek. Finally, we discuss how *distributed streams are ensured to terminate by the Stream Termination Protocol.*
- **Chapter 3** introduced the meta-level architecture for stream DSLs, called μCreek. It consists of two parts, μCreek_R and μCreek_C. μCreek_C is *a*

compile-time intercession framework that represents the stream as a stream of events at compile-time and allows the structure of the stream to be changed. μCREEK$_R$ is the run-time counterpart and *allows the semantics of the stream to be changed at run-time.* **Both μCREEK$_R$ and μCREEK$_C$ meta-programs are expressed as streams. In other words, the application and the meta-level are both expressed in the stream paradigm.** We evaluate the expressiveness of μCREEK by implementing operator fusion, timestamping, and parallelization of streams. We also evaluate the performance overhead of the meta-level. We concluded the chapter with a debugger implemented at the meta-level.

- **Chapter 4** introduced POTATO, a framework for Cyber-Physical System (CPS) applications. Potato is centered around stream-based programming. **POTATO addresses the challenges distilled from literature in a single programming paradigm, stream programming.** The hypothesis was that stream programming is ideal for expressing data-driven large-scale systems. This chapter shows how to tackle the software design challenges in CPS in the stream paradigm. First, the dynamic network is exposed to the application as a stream of join and leave events. Applications that use the *world stream* as a source to express applications as declarative transformations of this stream *deal with the dynamic network by design.* A device identifies itself on the network with a node descriptor that contains meta-information about the device. *The node descriptor allows devices to be designated based on logical properties, such as exposed services and type.* Devices can be updated via their *update stream*, a stream dedicated to transmit programs across the network. Devices multicast data on the network via their *data stream*, and can set up *dedicated communication channels using first-class programs.* We evaluate POTATO by implementing the "VUB 4.0" use case and comparing it to an implementation in a general purpose programming language without abstractions designed for CPSs.

- **Chapter 5** describes our implementation of POTATO and CREEK in broad strokes. We explain how the CREEK DSL creates Directed Acyclic Graphs (DAGs) at compile-time, and how DAGs are turned into streams. Next, it explains how the Stream Termination Protocol is implemented using links and monitors. Finally, the implementation of POTATO is explained.

6.3 Revisiting The Contributions

In chapter 1 we listed five challenges that programmers face when designing CPS applications today. In what follows we revisit our contributions and summarize how they address these challenges.

6.3.1 The Creek & μCreek Models

In chapter 2 we taxonomized the state of the art in stream DSLs, and designed a novel DSL, called Creek. Creek is used as an experimentation vehicle to implement a novel meta-level architecture for stream DSLs, called μCreek.

In a CPS there is heterogeneity between the devices connected to the network. The heterogeneity introduces a variety of *non-functional concerns* that the application logic has to address to integrate these devices in the network. As a result, the non-functional logic is entangled with the application logic. To the best of our knowledge there is no meta-level architecture design for stream-based languages.

μCreek is a meta-level architecture for stream DSLs founded on the canonical stream protocol (see section 2.2.5). μCreek consists of two complementary approaches: μCreek$_R$ and μCreek$_C$.

- μCreek$_C$ is **stream-based structural reflection and allows a meta-program to change the structure of a stream at compile-time**. This makes it possible to implement non-functional concerns such as operator fusion, time stamping, and encryption.
- μCreek$_R$ is **stream-based behavioral reflection and allows a meta-program to change the semantics of the stream at runtime**. μCreek$_R$ makes it possible to change the propagation semantics of a stream, and change the way an operator handles incoming and outgoing values.

In summary, **Creek and μCreek express the application logic, compile-time meta-programs, and run-time meta-programs in a single paradigm: streams.** In section 4.5 we show how μCreek can be used to integrate different types of thermometers into a single application.

6.3.2 The Potato Framework

In chapter 4 we introduced our design for a stream-based CPS framework. We explained our design, and validated it by implementing a use case from literature.

CPSs are event-driven systems that contain of heterogeneous devices connected via an open and unreliable network. Some devices in a CPS have limited computational power, and require other devices to execute complex computations. Our hypothesis was that **stream programming is an interesting foundation to build stream-based applications**, and we investigated how to address the software design issues in CPSs in the stream paradigm. Potato addresses the issues listed in chapter 1 as follows.

- **Challenge 2: Large Scale** The large-scale of the network is addressed by the way applications are defined in Potato. The *world stream* addresses the scalability because idiomatic applications define domain logic as distributed streams that are set up when the necessary devices are discovered on the network. As a result, the application is not affected if it works for a few devices or hundreds of devices.
- **Challenge 3: Unreliable Systems** The unreliable nature of the CPS its network can cause network partitions that affect the application. Potato adopts the Erlang approach of "let it crash". When a network partition occurs, the stream termination protocol (see section 2.3) ensures that the distributed streams are cleaned up on the network. In combination with the *'world stream*, the distributed streams are redeployed when the necessary devices are discovered.
- **Challenge 4: Open Systems** The open network in Potato is exposed to the application as the *world stream*, a stream that emits messages when the network topology changes. Canonical Potato applications are expressed in terms of the *world stream*, ensuring that they deal with the open network by design.
- **Challenge 5: Limited Computational Power** Some devices in a CPS have limited computational power, which limits the types of services they can provide to the network. Every Potato device has a unique *update stream*, which can be used by other devices to stream *first-class reactive programs* to the devices. Devices automatically deploy these applications locally. First-class reactive programs can be used to set up distributed streams, to involve more powerful devives in computations.

6.4 Limitations and Future Work

In this section we discuss the limitations of the work we presented.

Security Potato allows devices to distribute application logic across devices when they connect to the network. An obvious limitation of this approach is security. Malicious actors can join the network and inject malicious code onto devices.

Potato does not give a device control over which streams are deployed locally by other devices. Conversely, there is no limitation on which devices a device can deploy streams onto.

A possible solution to this problem is a form of capabilities per device. Cuttlefish [77] and MundoCore[6], for example, define the capabilities per device and only deploy services onto the devices with the right capabilities. These deployments are managed by the runtime and not at the application level. Devices in Potato define their capabilities through their node descriptor. When a device wishes to deploy a stream onto another device, it must designate it by filtering it out of the *world stream* based on its node descriptor. A device is free to designate any device it wants. Introducing automatic authorization when programs are deployed onto devices, or a form of code signing, can help in limiting which devices can be used to deploy code onto.

Code Management A deployment in Potato is a so-called "fire and forget" deployment. The application is transmitted to the target device and deployed there. The sender has no control or information about the deployment once the program has been transmitted. The receiver of the deployment has no control over the deployment either. This is particularly limiting in case of network failures. The current semantics of a network partition in distributed streams is that the streams are terminated. When the devices reconnect, the stream is recreated completely. This is wasteful in terms of network resources, but also in terms of computational power.

A first possible solution has been investigated by a master student. The approach is two-pronged. First, a first-class program can be annotated with instructions on the desired behavior in the face of network partitions. Possible behaviors are pausing the program, deleting the program, or keeping the program alive until the device reconnects. Second, when a program is deployed, a meta-stream is returned that streams information about the deployed program to the sender of the program. This meta-stream can be used by the Potato runtime to control the remote deployment.

Composable μCreek$_R$ Creek injects meta-behavior into all the streams defined in a module when a `behavior` pragma is present in that module. The compiler pairs this behavior to the Directed Acyclic Graphs (DAGs) at compile-time, and the runtime injects the behavior every time the DAG is deployed. A limitation of the current approach is that only one meta-DAG can be defined per module, and per stream. It is, for example, not possible to have pull-based semantics together with encryption semantics in a single stream without manually creating a specific behavior that implements both these concerns. The root of the problem is that these meta-behaviors are not serially composable. Directing the output (i.e., meta-responses) of the first behavior to the input (i.e., meta-messages) of the second behavior is not possible, because a μCreek$_R$ application has two distinct stages. First, the base stage calls the base level, and secondly, the effects stage sends the necessary messages. The composition of two meta-programs requires the base stages to be combined, and then the effects stages.

We see a possible starting point in splitting the two stages explicitly in the meta-program. This would allow the compiler to compose the stages freely, but would make the meta-program definition less elegant. The current structure of a meta-program does not require a clear separation of these two stages.

Higher-Order Streams As a consequence of the lack of composability of meta-streams, Creek does not support higher-order streams (i.e., runtime topology changes). Higher-order streams make it possible to add or remove operators in a stream at runtime. Currently this is only possible for sinks and sources. Consider the sample in section 6.4 as a hypothetical Creek program. For each value from the `source`, the `map` creates a new stream source, resulting in a stream of streams using the `Creek.Source.single` function that turns a single value into a stream that emits that value and then completes. The `flatten` operator is an operator that assumes its upstream to emit streams. Each of these streams is added as an upstream of the `flatten` operator. The values emitted by its upstreams are passed onto its downstream. This changes the topology of the stream at runtime.

```
source
~> map(fn value -> Creek.Source.single(value) end)
~> flatten
~> sink
```

Figure 6.1 graphically depicts this process. The `source` operator emits integers, and the `map` operator turns each of them into a `single`stream. The `flatten` operator deploys the streams at runtime and places itself as the sink of the deployed stream. In this particular case this results in a stream of integers again.

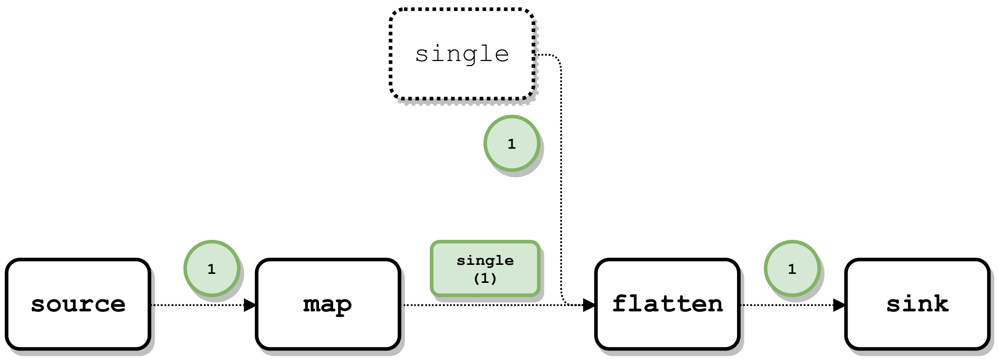


Figure 6.1: *A graphical depiction of the `flatten` operator.*

Meta-level Streams and Higher-Order Streams Consider the case where the stream in section 6.4 is deployed with a meta-behavior. Each stream that is added at runtime must understand the meta-level protocol of the original stream. However, if the new streams have a different meta-behavior it has to be composed with the existing meta behavior.

Quality of Service Potato proposes a design that integrates the network, devices, and their datastreams in the stream paradigm. This approach tackles challenges related to software design, but does not fully address concerns in context of Quality of Service. There are many approaches to add QoS to a distributed system. Some approaches require information about the underlying systems, such as CPU utilization, network congestion, and memory usage. However, from within *μ*Creek, it is only possible to offer QoS approaches that are based on information of the data itself, or the frequency of tranmission. In section 3.4.2 we explained what information is reified at the meta-level, and this information does not contain any introspection into the underlying hardware, or network connection. However, approaches such as only transmitting values that are significantly different from previous emitted values, throttling transmission, or caching values [56] are within the possibilities of *μ*Creek. Every meta-stream in *μ*Creek is stateful, and can inspect the application-level values at run-time and apply statistical models on transmission rate.

Real-Time Some CPS applications require some form of real-time constraints, such as hard real-time or soft real-time. Timing compliance is especially crucial for mission-critical systems, such as healthcare or aviation systems. Previous work has investigated how real-time compliance can be obtained in mobile agents for Cyber Physical Systems (CPSs) [23], Real Time Operating Systems (RTOS) [22], and reflective middlewares for embedded systems [29]. Work in context of functional reactive programming has also investigated how real-time constraints can be obtained in functional reactive programming [95], by statically defining the total cost of expressions in time and space. Meta-level approaches to express real-time constraints in a system have been done in context of actors [47] and operating systems [98]. Central to real time constraints based on reflection, is that the reflective systems need to have access to metrics about the underlying system such as the CPU, memory, and network.

*μ*Creek is a meta-level about the data streams between systems, and not about the hardware the application is deployed on. Therefore, *μ*Creek does not reify information of the underlying system to make real time constraints possible through reflection. Potato is built on top of the Erlang VM, which at best can guarantee soft real-time constraints. Creek and Potato do not have the facilities to improve upon this.

6.5 Closing Remarks

CPSs have introduced challenges in several fields such as network engineering, hardware engineering, and software engineering. We investigated on how to improve the state of the art in software engineering for CPSs. We proposed stream-based programming as an ideal starting point to express CPS applications, because it is well-suited to express event-driven systems. However, CPSs introduce software design challenges due to heterogeneous devices, dynamic networks, and unreliability of devices. We investigated how we can address these challenges from within the stream paradigm. Additionally, the heterogeneity of the devices requires a structural approach to separate the non-functional concerns from the application. **This dissertation proposed the first meta-level architecture for stream-based languages in a uniform programming paradigm, i.e., the stream paradigm.** Additionally, **we proposed a framework for CPS applications that exposes the dynamic network, the data exchange between devices, and the code deployments as streams.** To the best of our knowledge, Potato and Creek are the first stream-based approach that expresses all concerns of a CPS in the stream paradigm, and we consider our work a first step towards stream-based CPSs.

Appendix A

A Primer on Elixir Programming

The approach of this dissertation is demonstrated using two prototypes written in Elixir. Creek, an embedded Domain Specific Language (DSL), and Potato, a programming framework for Cyber Physical Systems (CPSs). To ensure this dissertation is self-contained, this chapter serves as a primer on Elixir and will be referenced in this dissertation.

Figure A.1: *Joe Armstrong, inventor of Erlang. Image used with permission from learnyousomeerlang.com.*

Elixir is an industry-strength language invented by Jose Valim in 2011. The language is built on top of the Erlang programming language and reuses much of its ecosystem (i.e., virtual machine, compiler, and standard library). Elixir and Erlang are dynamically typed, functional, declarative, concurrent, and distributed programming languages. Everything discussed in this chapter applies to both languages unless explicitly stated otherwise. Originally, Erlang was designed to program telephone systems. Through the years, however, together with Elixir, it has evolved as one of the best choices to create communication-driven software (90% of all Internet traffic goes through Erlang-programmed hardware [1]). Companies such as Pinterest, The Financial Times, Toyota, Discord, and Pepsi have redesigned large-scale systems from more traditional languages to Elixir because of the advantages it offers.

[1] `https://codesync.global/media/https-youtu-be-077-xjv6plq/`

We have chosen to implement prototypes of our approach in the Elixir ecosystem for the following reasons. Elixir takes over the majority of features present in Erlang. Erlang is a good candidate because it is highly concurrent, has built-in mechanisms for fault-tolerance in distributed systems, and offers live code updates. Elixir extends Erlang with a more modern syntax, extensions to the Erlang standard library, and improved meta-programming facilities. The combination of all these features makes Elixir an ideal candidate to develop distributed applications on open networks, such as CPSs.

This chapter gives a short introduction to the Elixir programming language and the features we rely on in our prototypes (i.e., Creek and Potato).

Sequential Elixir

Primitive Data Types

Elixir has eight primitive data types.

- **Integers** Integers are represented using *Arbitrary-precision arithmetic*, meaning that there is no minimum and maximum value, and they have exact precision. Examples are `123` and `-123`.
- **Atoms** Atoms are alphanumerical constants whose value is their name. Two atoms with the same value are always equal. Examples are `:foo` and `:"an atom"`.
- **Floats** Floating-point numbers in Elixir are represented as IEEE 754 [1] numbers; they are represented as 64-bit numbers. Examples are `1.23` and `123.4e10`.
- **Pids** Process identifiers are the unique identifier for an Elixir process. A *local pid* is unique within the virtual machine instance, and a *global pid* is universally unique. Pids cannot be manually created; only the `spawn` primitive can create pids. An example is `#PID<0.112.0>`.
- **Functions** Functions are lexically scoped closures. Free variables are not allowed, and functions can be nested. Functions are created using the `fn x -> x + 1 end` syntax, or the shorthand `&(&1 + 1)`.
- **Strings** Strings are syntactic sugar to create a binary representation of a UTF-8 encoded string. Examples are `"Elixir is fun"` and `"héllo"`.
- **Tuples** Tuples are the primary compound data type used in Elixir. They have a fixed number of items and are immutable. They are created using the `{x, y}` syntax. Each element in a tuple can be a primary or compound

value. Values are accessed in constant time. Examples are `{:tag, 123.4}` and `{:tag, {1,2}}`.

- **Lists** Lists are dynamic containers for a variable amount of values. A list is created using the `[x | xs]` syntax, where `x` is a value, and `xs` a list of values. `[]` is the empty list. The list `[1 | [2 | []]]` can be written using the shorthand `[1,2]`. Lists are accessed in linear time.

Variables

Variables in Elixir are alphanumerical tokens that start with a lowercase letter and contain alphanumerical characters. `x1` and `true?` are valid variable names. Elixir variables are *single assignment*. When a variable is *bound* to a value, it cannot be changed. Opposed to Erlang, however, Elixir variables can be shadowed. The example below shows how assignment in Erlang is simulated using new variable names and how it is done Elixir by shadowing. Ignoring the capitalization of the variables, the Elixir program on the right is not a valid Erlang program, but the Erlang program is a valid Elixir program.

`X = 5`
`X1 = X + 1`
	`x = 5`
	`x = x + 1`

Table A.1: *Single Assignment in Erlang (left) and rebinding in Elixir (right).*

Pattern Matching

Elixir and Erlang rely heavily on pattern matching for function definitions, composite data structure deconstruction, and conditionals. A pattern is defined as a data type, a variable, or constructed with lists and tuples of patterns. A *ground term* in Elixir is a term where no variables are present. It can be a primitive data type, tuple, or list. Pattern matching is the process of unifying ground terms with patterns. Patterns can contain the same variable more than once, requiring them to be unified with the same ground term. Listing 58 shows three examples of pattern matching. The right-hand side of the expression is a ground term, and the left-hand side is a pattern.

```
iex(1)> {name, born} = {"Joe Armstrong", {:born, 1950}}
iex(2)> {:born, year} = born
iex(3)> age = 2021 - year
```

Listing 58: *REPL session to showcase pattern matching.*

Modules and Functions

Elixir code is divided into modules.

A module is defined using the `defmodule <name>` syntax. Any module consists of functions or module attributes.

A function is defined using the `def <name>(<params>)` syntax. A function defined with `defp <name>(<params>)` is a private function and can only be accessed from within its module. Function definitions cannot be nested. Functions can be overloaded in function arity. In Elixir terminology, each overloaded function is called a *clause*. Elixir has *tail-call optimization*; function-calls in tail position do not create an additional stackframe. Module attributes are compile-time constants. They are defined using the `@<name> <value>` syntax and can only be referenced from the module in which they are defined.

The example in listing 59 defines the `Math` module. The module implements the `factorial` function. In the example, the `factorial` function is overloaded with an accumulator parameter to allow for tail-call optimization by the interpreter. The version of the module is defined as the `version` module attributed with value 1.0.

```
defmodule Math do
    @version 1.0
    def factorial(n), do: factorial(n, 1)
    defp factorial(0, acc), do: acc
    defp factorial(n, acc), do: factorial(n - 1, acc * n)
end
```

Listing 59: *Iterative factorial implementation in Elixir.*

```
iex(1)> Math.factorial(5)
120
```

Listing 60: *REPL session to calculate the factorial of 5.*

Functions in Elixir are *first-class citizens*. Functions can be assigned to variables, returned from functions, and passed to functions. To capture a function in a variable the `&Module.function/arity` syntax is used. For example, `&Math.factorial/1` captures the factorial function from the `Math` module. Functions assigned to variables are called with a slightly different syntax. Regular functions are called by their names, followed by parentheses' arguments. First-class functions require a dot between the name and the parentheses. Listing 61 show a function call to a higher-order function.

```
iex(1)> fac = &Math.factorial/1
iex(2)> fac.(5)
120
```

Listing 61: *REPL session to calculate the factorial of 5 using higher-order functions.*

Concurrent Elixir

Processes

Elixir implements the actor model, meaning that the basic unit of concurrency is an actor. In Elixir, actors are called *processes*. The `spawn` function creates a process by running its argument in a new process. The function expects an anonymous function as its argument. An overloaded version takes a module, function name, and list of parameters. Listing 64 call the same function twice using both clauses of the `spawn` function. Processes in Elixir are lightweight in terms of memory and computational overhead. It is not uncommon to have thousands of processes running in a single application. The time it takes to create a process is in the order of microseconds.

```
iex(1)> pid = spawn(fn -> SQL.connect("127.0.0.1", "admin", "secret") end)
iex(2)> pid = spawn(SQL, :connect, ["127.0.0.1", "admin", "secret"])
```

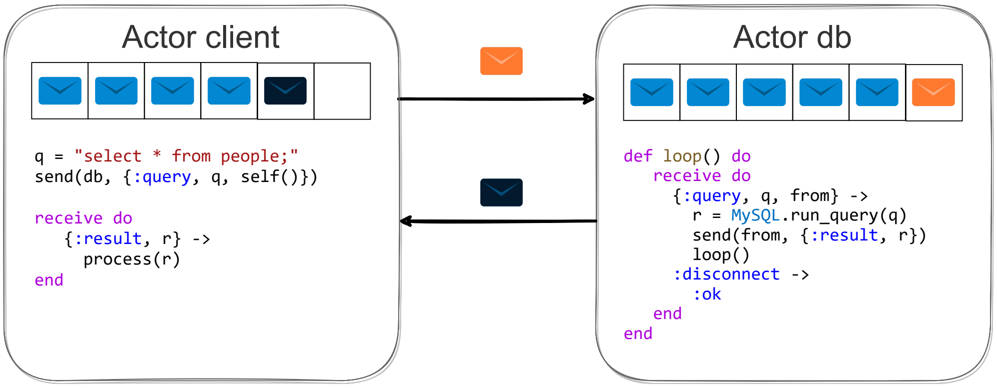
Listing 62: *REPL session to create a database process in a separate process.*

Processes can communicate with each other through message passing. A process has a *mailbox* in which messages are queued. A message can be sent to a given process with the `send` function. It takes a pid as its first argument (the recipient) and any Elixir term as its second argument (the message). A process can obtain its process identifier using the `self` function. Messages are queued into the actor's mailbox in the order in which they were sent. If A sends five messages to B, the messages are guaranteed to arrive in the order they were sent.

An actor can process a message from its inbox with the `receive` statement. The `receive` statement defines one or more patterns. The first message in the inbox is tested against each pattern. If a pattern matches, the sequence of statements after the pattern is executed. No additional messages are processed until the next receive statement is encountered. If no pattern matches any message in the inbox, the receive statement blocks until a matching message arrives. An optional `after` defines a timeout. If no matching message is received within the defined interval, the receive block executes the expression sequence after the `after` clause.

```
receive do
  pattern_1 ->
      exp_1
      ..
      exp_n
  pattern_n ->
      exp_1
      ..
      exp_n
  after timeout ->
      exp_1
      ..
      exp_n
end
```

Listing 63: *Syntax of the* `receive` *statement.*



```
q = "select * from people;"
send(db, {:query, q, self()})

receive do
   {:result, r} ->
     process(r)
end
```

```
def loop() do
   receive do
     {:query, q, from} ->
       r = MySQL.run_query(q)
       send(from, {:result, r})
       loop()
     :disconnect ->
       :ok
   end
end
```

Figure A.2: *Two actors sending a message to each other.*

Links and Monitors

The Erlang mantra is *"let it crash"*. In Erlang and Elixir, applications are not supposed to try and catch all the possible exceptional states to avoid application crashes. It is idiomatic not to catch exceptions and let a process crash. Elixir offers mechanisms for monitoring processes and reacting to a crashed process.

Process Monitors When a process is monitored, the monitoring process will receive a message in case the monitored process crashes. The message allows the monitor to respawn the process, for example. To monitor a given process, the `monitor` function is used. For example, if `pid` is a variable containing a process, it can be monitored using `Process.monitor(pid)`. If the monitored process crashes, a message containing the error is sent to the monitoring process.

For instance, if the `pid` process encountered a division by zero, the following message will be in the mailbox of the monitoring process.

```
{:DOWN, #Reference<0.3246560077.66060294.216107>,
        :process, #PID<0.17167.0>,
        {:badarith, [{:erlang, :/, [1, 0], []}]}}
```

Listing 64: *Example of a monitor message. The monitored process encountered a division by zero.*

Process Links A process link is a strong bond between two processes than monitors. When one of two linked processes fails, the other process is brought down with it. For instance, a process that acts as an interface to a database and a process that manages the database connection should be linked together. It is easier to bring both of them down in case of failure and bring up two fresh instances. A process can link to many processes and can monitor many processes. A process can be monitored by many processes and linked to many processes. Multiple processes that are linked form a cluster; in case of failure, all processes terminate.

To link to a pid `pid` the `Process.link(pid)` function is used.

Distributed Elixir

Elixir has features to distribute computation across several virtual machine instances. A single instance of the Erlang Virtual Machine is called a *node*. A single computer can run multiple Erlang nodes. Every computer has to run a single instance of the `Erlang Port Mapper Daemon` (EPMD). This process keeps track of the process identifiers on its own machine. It communicates with other machines to track which machine contains which processes.

If a process wants to communicate with a node on another server, it will ask the local server's EPMD for the routing information. Regardless of the virtual machine (i.e., same computer or remote server), the EPDM will supply the routing information and ensure that the messages get to their destination.

In Erlang, it is possible to spawn a function on a remote node in the network. Instead of passing a pid to the `spawn` function, a node name can be passed (e.g., `:"othermachine@1.2.3.4"`)). Spawning on a remote node transports the function across the network and executes it there. This mechanism is limited in the code that can be sent, however. When the function references code that is defined in a non-built-in module, e.g., our `Math` module, the code is expected to reside on the remote machine, or it will throw an error. Code is not

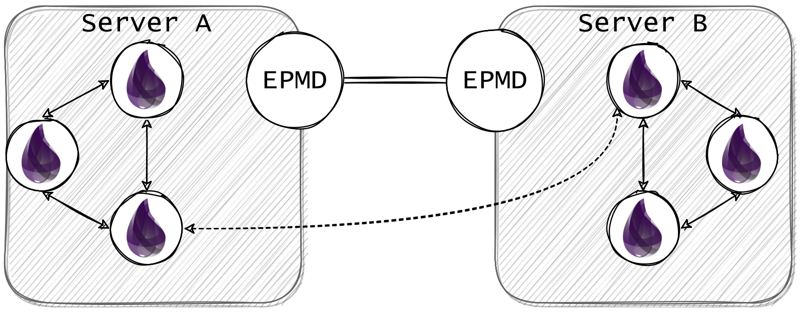


Figure A.3: *Diagram of communication between two physical servers, each running three nodes.*

automatically transmitted when a function is executed on a remote machine. Erlang has a built-in code server that allows devices to exchange code, but this has to be manually enabled.

The Erlang Actor Model

The actor model, invented by Hewitt et al. [52], knows many variations [63]. The variation present in Erlang and Elixir is the process model. Each actor is modeled as a lightweight process that runs from start to finish. Each process has its private memory and message mailbox. During execution, a process can do arbitrary function calls and block its execution to wait for a message using the **`receive`** statement. If no message in the inbox matches any of the patterns, the process blocks.

A **`receive`** statement can have multiple clauses. Patterns are matched from first to last. The first pattern that matches any message in the inbox is executed.

We have previously mentioned that Elixir has tail-call optimization. Tail-call optimization makes it possible to create infinite processes using recursive functions. The process is contained in a function that recursively calls itself and keeps its state in its arguments. The `server/2` function in the `EchoServer` module in listing 65 is an example of such an idiomatic Elixir process. The `server` process keeps its state in the `count` variable which is updated every recursive call. To run this function in a separate process, `spawn/1` must be used. A new server can be started using the `spawn(&EchoServer.start/0)` expression.

```
defmodule EchoServer do
  def server(count) do
    receive do
      {m, from} ->
        send(from, "#{m} was the #{count}th message.")
        server(count + 1)
    end
  end

  def start() do
    server(0)
  end
end
```

Listing 65: *An echo server that echos the sent messages and keeps track of how many messages it received.*

Macros

Elixir has *meta-programming* in the form of macros. Meta-programming is "code about code". There are many forms of meta-programming (see chapter 2), and macros are only one of them. Macros are code that generates code during the compilation of the program. This process is called *macro expansion*. Macros are useful to extend a programming language with new primitives, and therefore the primary tool to create Domain Specific Languages (DSLs) in Elixir. For instance, Elixir is a functional language and therefore does not contain imperative constructs such as a `while` loop. Using macros, it is possible to add a `while` loop as syntactic sugar over a recursive function.

Macros are functions that take code as parameters and return code as a result during compilation. To make code a first-class citizen of the programming language, it has to be represented as a value. Elixir represents code as the AST of the code. The AST is defined using primitive data types such as maps and lists. For example, the expression `1 + 2` its AST is `{:+, [context: Elixir, import: Kernel], [1, 2]}`. Every macro takes an AST as its parameter and must return an AST.

To turn any Elixir expression into its AST representation, the `quote` function must be used. The parameter of a quote expression is the Elixir code that will be turned into its AST representation. For example, `quote do: 1 + 2` returns `{:+, [context: Elixir, import: Kernel], [1, 2]}`. When defining an AST (i.e., in a `quote` block), other ASTs can be injected using the `unquote` function.

Listing 66 defines the `while` macro. A macro is defined using the `defmacro`

```
defmacro while(predicate, do: body) do
  quote do
    loop = fn loop ->
      if unquote(predicate) do
        unquote(body)
        loop.(loop)
      end
    end
    loop.(loop)
  end
end
```

Listing 66: *A macro that adds a `while` loop to Elixir.*

```
def wait_on_termination(pid, t) do
    while Process.alive?(pid) do
        Process.sleep(100)
    end
end
```

Listing 67: *A function that uses the `while` macro. The function blocks until the given pid terminates.*

syntax and takes an arbitrary amount of parameters. Each of these parameters is an AST. Inside the macro, a `quote` block is opened to define the AST that will be returned. The macro transforms the `while` expression into an anonymous function that is called with itself as an argument to simulate recursion. Listing 67 shows a function that employs the `while` macro to wait for a process to terminate.

In the definition of the `while` macro variable names such as `loop` are used. To avoid these names clashing with names in the application, Elixir ensures macros are *hygienic*. Behind the scenes, Elixir will ensure that these variable names are unique and never shadow bindings from outside the macro. The macro programmer can override hygiene with the `var!` function to introduce variables in a `quote` block.

Conclusion

In this chapter, we introduced the Elixir programming language. We started with the sequential parts, followed by the local concurrency and distributed aspects. Finally, we gave a brief overview of macros in Elixir.

After presenting the features present in Elixir to design concurrent and distributed systems, as well as macro facilities, we conclude it is a good fit to develop a DSL and programming framework for Cyber-Physical System (CPS). Languages that do not have these features would require us to implement these features ourselves.

Appendix B

Source Code Logging

```
defmodule Logging do
  use Creek.MetaBehaviour

  dag default as base ~> effects

  dag snk_default as filter(&(not match?({_, :next, _, _}, &1)))
                        ~> default

  dag src_default as filter(&(not match?({p, :tick}, &1)))
                        ~> default

  dag opr_default as filter(&(not match?({_, :next, _, _}, &1)))
                        ~> default

  # Source
  dag log_next as filter(&match?({_, :next, _, _}, &1))
                     ~> map(fn {p, :next, v, f} ->
                       IO.puts("#{p.node.name} incoming: #{v} from #{f}")
                       {p, :next, v, f}
                     end)
                     ~> base
                     ~> map(fn result ->
                       case result do
                         {p, {state, :next, value}} ->
                           IO.puts("#{p.node.name} outgoing: #{inspect(value)}")

                         _ ->
                           :ok
                       end

                       result
                     end)
                     ~> effects

  # Source
  dag source_tick as filter(&match?({p, :tick}, &1))
                        ~> map(fn {p, :tick} ->
                                    IO.puts("Source signaled to produce!")
                                    {p, :tick}
                                  end)
                                  ~> base
                                  ~> map(fn result ->
                                    case result do
                                      {p, {state, :tick, value}} ->
                                        IO.puts("Source outgoing: #{inspect(value)}")

                                      _ ->
                                        :ok
                                    end

                                    result
                                  end)
                                  ~> effects

  defdag operator(src, snk) do
    src
    ~> dup()
    ~> (log_next ||| opr_default)
    ~> merge()
    ~> snk
  end

  defdag source(src, snk) do
    src
    ~> dup()
    ~> (source_tick ||| src_default)
    ~> merge()
    ~> snk
  end

  defdag sink(src, snk) do
    src
    ~> dup()
    ~> (log_next ||| snk_default)
    ~> merge()
    ~> snk
  end
end
```

Listing 68: *Full source code for logging in μ*C<span>REEK</span>$_R$.

Appendix C

Source Code Operator Fusion in μCreek$_C$

```
defmodule Fusion do
  use Structural

  defdag metadag(src, snk) do
    src
    ~> dup()
    ~> (edge ||| others)
    ~> merge()
    ~> proceed
    ~> snk
  end

  defdag edge as
    filter(fn event ->
      match?({{:edge, _, _, _, _}, _, _}, event)
      end)
    ~> map(fn {{:edge, from, fidx, to, toidx}, dag, it} ->
      a = fetch!(dag, from)
      b = fetch!(dag, to)
      case {a.name, b.name} do
        {"map", "map"} ->
          [x] = inputs(dag, a)
          c = fuse(a, b)
          dag = delete(dag, a)
          dag = delete(dag, b)
          dag = add!(dag, c)
          {{:edge, x.ref, 0, c.ref, 0}, dag, it}
        _ ->
          {{:edge, from, fidx, to, toidx}, dag, it}
      end
    end)

  defdag others as
    filter(fn event ->
      not match?({{:edge, _, _, _, _}, _, _}, event)
    end)
end
```

Listing 69: *Full source code for operator fusion in μCreek$_C$.*

Appendix D

Source Code Parallelization in μCreek$_C$

```
defmodule Parallelize do
  use Structural

  defdag op as,
    filter(fn event ->
      match?({{:operator, _}, _, _}, event)
    end)
    ~> map(fn {{:operator, op}, dag, it} ->
      if Keyword.has_key?(op.opts, :parallel) do
        factor = Keyword.get(op.opts, :parallel)

        #    +----+
        #    |dup|
        #    +----+
        #    v   v
        # +--++ ++--+
        # |map| |map|
        # +----+ +----+
        #      v v
        #    ++-++
        #    |mrg|
        #    +----+
        # The output of the transform must be duplicated factor times.
        # After the operations it must also be merged together.

        # The transform operator will tag each value with an index.
        # This is used to balance them across the parallel pipelines.
        f1 =
          Creek.Operator.transform(
            0,
            fn x, state ->
              tag = rem(state + 1, factor)
              {tag, {tag, x}}
            end
          )

        insert(f1)

        # The parallel pieplines need to be merged together in the end.
        f_n = Creek.Operator.merge(factor, start: f1.ref)
        insert(f_n)

        # The transformed tagged values need to duplicated across
        # each pipeline using duplicate.
        f2 = Creek.Operator.dup(factor)
        insert(f2)

        # Transform emits to duplicate directly.
        connect(f1, 0, f2, 0)

        dag =
          0..(factor - 1)
          |> Enum.reduce(dag, fn i, dag ->
            f =
              Creek.Operator.filter(fn {tag, v} ->
                tag == i
              end)

            m =
              Creek.Operator.map(fn {_tag, value} ->
                op.arg.(value)
              end)

            insert(f)
            insert(m)
            # Connect map after filter.
            connect(f, 0, m, 0)
            # Each map outputs to the merge.
            connect(m, 0, f_n, i)
            # Each filter is connected to the duplicator.
            connect(f2, i, f, 0)
            dag
          end)

        {{:operator, f_n}, dag, it}
      else
        {{:operator, op}, dag, it}
      end
    end)

  defdag edge as,
    filter(fn event ->
      match?({{:edge, _, _, _, _}, _, _}, event)
    end)
    ~> map(fn {{:edge, from, fidx, to, toidx}, dag, it} ->
      b = fetch!(to)

      if Keyword.has_key?(b.opts, :start) do
        actual_to = Keyword.get(b.opts, :start)
        {{:edge, from, fidx, actual_to, toidx}, dag, it}
      else
        {{:edge, from, fidx, to, toidx}, dag, it}
      end
    end)

  defdag metadag(src, snk) do
    src
    ~> dup(3)
    ~> (op ||| edge ||| default_name)
    ~> merge(3)
    ~> proceed()
    ~> snk
  end
end
```

Listing 70: *Full source code for parallelizing operators in μCreek$_C$.*

Appendix E

Source Code Pull Semantics in μCreek$_R$

```
defmodule SmartPull do
  use Creek.MetaBehaviour

  # This \gls{DAG} handles all events with default behavior.
  defdag default as base ~> effects

  ##############################################################################
  # Operators

  # This \gls{DAG} propagates every demand message upstream.

  defdag forward_demand as
        filter(&match?({_, :meta, :demand, from}, &1))
        ~> map(fn {p, :meta, :demand, from} ->
        # Demand as operator means it needs to be propagated.
        # Only propagate to the operators which are not marked
        # as "demanded".
        demanded = p.meta_state
        to_demand = p.us |> Enum.filter(&(not MapSet.member?(demanded, &1)))
        propagate_upstream_meta(:demand, to_demand, p.pid)
        meta_state = MapSet.new(p.us)
        {%{p | meta_state: meta_state}, :ok}
        end)

  # If an operator does not propagate a vlaue the demand is "lost".
  # As soon as no value is propagated in response, a new demand is sent.
  defdag opr_next as
        filter(&match?({_, :next, _, _}, &1))
        ~> map(fn {p, :next, v, from} ->
            meta_state = p.meta_state |> MapSet.delete(from)
            {%{p | meta_state: meta_state}, :next, v, from}
        end)
        ~> base()
        ~> map(fn {p, base_response} ->
            if match?({_, :skip}, base_response) do
            demanded = p.meta_state
            to_demand = p.us |> Enum.filter(&(not MapSet.member?(demanded, &1)))
            propagate_upstream_meta(:demand, to_demand, p.pid)
            end

            {p, base_response}
        end)
        ~> effects()

  defdag opr_default as
        filter(&(not match?({_, :meta, :demand, _}, &1)))
        ~> filter(&(not match?({_, :next, _, _}, &1)))
        ~> filter(&(not match?({_, :init_opr}, &1)))
        ~> default

  defdag init_opr as
        filter(&match?({_, :init_opr}, &1))
        ~> base()
        ~> map(fn {p, resp} ->
            p = %{p | meta_state: MapSet.new()}
            {p, resp}
        end)
        ~> effects()

  defdag operator(src, snk) do
    src
    ~> dup(4)
    ~> (opr_default ||| forward_demand ||| opr_next ||| init_opr)
    ~> merge(4)
    ~> snk
  end

  ##############################################################################
  # Sources
  # Intercept the init event for sources to stop them from ticking themselves.
  defdag init_src as
        filter(&match?({_, :init_src}, &1))
        ~> base()
        ~> map(fn {p, {state, :initialized}} ->
            # Here we would normally send tick to ourselves, but we dont (pull).
            {%{p | state: state}, :ok}
        end)

  # This \gls{DAG} handles the demand messages.
  defdag demand_src as
        filter(&match?({p, :meta, :demand, _}, &1))
        ~> map(fn {p, :meta, :demand, _} ->
        # If a source receives demand it ticks itself.
        send_self({:tick}, p.pid)
        {p, :ok}
        end)

  # If a source gets a tick event (from itself)
  # it will produce a value and tick itself again.
  # We intercept that tick and stop from sending it.
  defdag tick_src as
        filter(&match?({p, :tick}, &1))
        ~> base()
        ~> map(fn base_result ->
            case base_result do
            {p, {state, :complete}} ->
                effects_complete(nil, p.ds, p.us, p.pid)
                {%{p | state: state}, :ok}

            {p, {state, :tick, value}} ->
                propagate_downstream({:next, value}, p.ds, p.pid)
                {%{p | state: state}, :ok}
            end
        end)

  # This \gls{DAG} handles all events except the ones we intercepted.
  defdag src_default as
        filter(&(not match?({_, :init_src}, &1)))
        ~> filter(&(not match?({_, :meta, _, _}, &1)))
        ~> filter(&(not match?({p, :tick}, &1)))
        ~> default

  defdag source(src, snk) do
    src
    ~> dup(4)
    ~> (src_default ||| init_src ||| demand_src ||| tick_src)
    ~> merge(4)
    ~> snk
  end

  ###############################################################################
  # Sinks

  # When a sink is initialized it normally doesnt do anything.
  # In pull-based we must send the first pull message.
  defdag init_snk as
        filter(&match?({_, :init_snk}, &1))
        ~> base()
        ~> map(fn {p, {_, :ok}} ->
            # Normally no side-effects happen in a sink init,
            # but now e must propagate demand upstream.
            propagate_upstream_meta(:demand, p.us, p.pid)
            {p, :ok}
        end)

  # This \gls{DAG} ensures that a new demand is sent when a next value arrived.
  defdag next_snk as
        filter(&match?({_, :next, _, _}, &1))
        ~> default()
        ~> map(fn {p, :ok} ->
            # After the default, we send demand upstream.
            propagate_upstream_meta(:demand, p.us, p.pid)
            {p, :ok}
        end)

  # This \gls{DAG} handles all events except the ones we intercepted.
  defdag snk_default as
        filter(&(not match?({_, :init_snk}, &1)))
        ~> filter(&(not match?({_, :next, _, _}, &1)))
        ~> default()

  defdag sink(src, snk) do
```

```
    src
    ~> dup(3)
    ~> (init_snk ||| snk_default ||| next_snk)
    ~> merge(3)
    ~> snk
  end
end
```

Listing 71: *Full source code for pull-based semantics in μCREEK$_R$.*

Appendix F

Decoding Thermometer Measurements in μCreek$_R$

```
defmodule ThermometerMeta do
  use Creek.MetaBehaviour

  def is_xml?({:xml, _}), do: true
  def is_xml?(_), do: false

  def is_json?({:json, _}), do: true
  def is_json?(_), do: false

  def xml_to_float({:xml, value}), do: value
  def json_to_float({:json, value}), do: value

  dag not_next as filter(&(not match?({_, :next, _, _}, &1)))
                 ~> base
                 ~> effects

  dag next as filter(&match?({_, :next, _, _}, &1))
                  ~> map(fn {state, :next, encoded, from} ->
                    decoded =
                      cond do
                        is_xml?(encoded) -> xml_to_float(encoded)
                        is_json?(encoded) -> json_to_float(encoded)
                        true -> encoded
                      end

                    {state, :next, decoded, from}
                  end)
                  ~> base()
                  ~> effects()

  dag encoding_meta as dup
                      ~> (next ||| not_next)
                      ~> merge

  defdag operator(src, snk) do
    src
    ~> base
    ~> effects
    ~> snk
  end

  defdag source(src, snk) do
    src
    ~> encoding_meta
    ~> snk
  end

  defdag sink(src, snk) do
    src
    ~> base
    ~> effects
    ~> snk
  end
end
```

Listing 72: *Full source to decode the values from thermometers.*

Appendix G

Source Code Anabranch Directed Acyclic Graph (DAG) in μCreek$_R$

```
defmodule Debugging2 do
  use Creek.MetaBehaviour

  dag default as base ~> effects

  ##############################################################################
  # Operators
  dag opr_outgoing as map(fn r ->
                      case r do
                        {p, {state, :next, value}, _} ->
                          {:outgoing, p.pid, state, value}

                        {p, {state, :ok}, _} ->
                          {:update, p.pid, state}

                        _ ->
                          :skip
                      end
                    end)
                    ~> filter(fn x -> x != :skip end)

  dag opr_complete as map(fn r = {p, {state, instr}, from} ->
                      if instr == :complete do
                        {:complete, p.pid}
                      else
                        :skip
                      end
                    end)

  dag incoming as map(fn {p, :next, v, from} ->
                    {:incoming, p.pid, v, p.state}
```

```
                  end)

  dag opr_done as map(fn r = {p, _} ->
                    if p.us == [] do
                      {:complete, p.pid}
                    else
                      :skip
                    end
                  end)
                  ~> filter(fn x -> x != :skip end)

  dag rest? as filter(&(not match?({_, :next, _, _}, &1)))
               ~> filter(&(not match?({_, :complete, _from}, &1)))
               ~> filter(&(not match?({_, :meta, _m, _from}, &1)))

  dag next?(as filter(&match?({_, :next, _, _}, &1)))

  dag complete?(as filter(&match?({_, :complete, from}, &1)))

  defdag operator(src, snk, sock) do
    let meta?(as filter(&match?({_, :meta, m, from}, &1)))

    # Update the argument if this is the right node.
    let metas as meta?
                 ~> map(fn {p, :meta, m, _from} ->
                  IO.inspect "Meta message in meta stream: #{inspect m}"
                   case {m, inspect(p.pid)} do
                     {{:update_arg, arg, pid}, pid} ->
                       {arg, _} = Code.eval_string(arg)
                       IO.inspect arg, label: "new arg"
                       {%{p | node: %{p.node | arg: arg}}, :ok}

                     _ ->
                       {p, :ok}
                   end
                 end)
                 ~> snk

    # Handle next events for operators.
    let incomings as incoming
                     ~> filter(fn x -> x != :skip end)
                     ~> sock

    let outgoings as opr_outgoing
                     ~> filter(fn x -> x != :skip end)
                     ~> sock

    let do_effects as effects
                      ~> snk

    let next as next?
                ~> dup(2)
                ~> (incomings |||
                      base
                      ~> dup
                      ~> (outgoings ||| do_effects))

    # All unprocessed events.
    let others as rest? ~> default ~> snk

  # Handle completes.
  let complete_before_effects as opr_complete
                                 ~> filter(fn x -> x != :skip end)
                                 ~> sock

  let complete_after_effects as effects
                                ~> dup
                                ~> (snk ||| opr_done ~> sock)

  let completes as complete?
                   ~> base
                   ~> dup
                   ~> (opr_complete
                       ~> sock |||
                         effects
                         ~> dup
                         ~> (snk ||| opr_done ~> sock))

  src
  ~> dup(4)
  ~> (next ||| others ||| completes ||| metas)
end

################################################################################
# Sources

defdag source(src, snk, sock) do
  # Filters.
  let tick? as filter(&match?({_, :tick}, &1))
  let rest? as filter(&(not match?({_, :tick}, &1)))

  # All other events.
  let rests(as rest? ~> default ~> snk)

  # Tick events.
  let export as map(fn e ->
                 case e do
                   {p, {state, :tick, value}} ->
                     {:outgoing, p.pid, state, value}

                   {p, {_, :complete}} ->
                     {:complete, p.pid}

                   _ ->
                     :skip
                 end
               end)
               ~> sock

  let ticks as tick?
               ~> base
               ~> dup
               ~> (export ||| effects ~> snk)

  src
  ~> dup
  ~> (ticks ||| rests)
end
```

```

  ##############################################################################
  # Sources

  defdag sink(src, snk, sock) do
    let next? as filter(&match?({_, :next, _, _}, &1))
    let rest? as filter(&(not match?({_, :next, _, _}, &1)))
    let complete? as filter(&match?({_, :complete, _}, &1))

    let rests as rest? ~> default ~> snk

    let incoming as map(fn {p, :next, v, _} ->
                      {:incoming, p.pid, v, p.state}
                    end)

    let incomings as incoming
                    ~> filter(fn x -> x != :skip end)
                    ~> sock

    let nexts as next?
                ~> dup(2)
                ~> (incomings |||
                      base
                      ~> effects
                      ~> snk)

    let export_completes as map(fn r = {p, {state, instr}, from} ->
                                  if instr == :complete do
                                    {:complete, p.pid}
                                  else
                                    :skip
                                  end
                                end)
                                ~> filter(fn x -> x != :skip end)

    let completes as complete?
                    ~> base
                    ~> dup
                    ~> (export_completes ~> filter(fn x -> x != :skip end) ~> sock |||
                          effects
                          ~> snk)

    src
    ~> dup(3)
    ~> (nexts ||| rests ||| completes)
  end
end
```

Listing 73: *Full source code for operator fusion in μCreek$_C$.*

Appendix H

Source Code for Potato Scenarios

Scenario 1: Adaptive Lecture Hall

```
defmodule HVAC do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  require Logger
  use Creek
  use Potato.DSL

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :vaillant,
      type: :hvac,
      name: "hvac",
      room: Keyword.get(opts, :room),
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run(room) do
    init(room: room)
  end

  def turn_on() do
    IO.puts("Turning on")
  end

  def turn_off() do
    IO.puts("Turning off")
  end
end
```

```elixir
defmodule Ventilation do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  require Logger
  use Creek
  use Potato.DSL

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :renson,
      type: :ventilation,
      room: Keyword.get(opts, :room),
      name: "ventilation",
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run(room) do
    init(room: room)
  end

  def set_ventilation(percentage) do
    IO.puts("Setting percentage to #{percentage}%.")
  end
end
```

```
defmodule Remus do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  alias Potato.Network.Observables, as: Net
  require Logger
  use Creek
  use Potato.DSL
  alias Creek.Source.Subject, as: Subject
  alias Creek.Source, as: Source

  def init() do
    # Our node descriptor.
    nd = %{
      hardware: :android,
      type: :phone,
      room: "room_3",
      name: "remus",
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  ##############################################################################
  # Turn off all plugs in current room.

  defdag disable_plugs(src, snk, room) do
    src
    ~> filter(fn {event, device} ->
      self = Potato.Network.Meta.get_local_nd()
      event == :join and device.room == self.room and device.type == :plug
    end)
    ~> map(fn {:join, plug} ->
      p =
        program do
          Plug.turn_off()
        end

      Subject.next(plug.deploy, p)
    end)
    ~> snk
  end

  ##############################################################################
  # Stream the temperature from all thermometers in the room.

  defdag control_hvac(src, snk, limit) do
    src
    ~> map(fn x ->
      IO.puts("HVAC got temperature: #{inspect(x)}")
      x
    end)
    ~> average()
    ~> map(fn t ->
      if t > limit do
        HVAC.turn_on()
      else
        HVAC.turn_off()
      end
    end)
    ~> snk
  end

  defdag update_hvac(src, snk, temp_source, desired) do
    src
  ~> filter(fn {event, device} ->
    self = Potato.Network.Meta.get_local_nd()
    event == :join and device.room == self.room and device.type == :hvac
  end)
  ~> map(fn {:join, hvac} ->
    p =
      program do
        deploy(control_hvac, src: temp_source, snk: Creek.Sink.ignore(nil), limit: desired)
        nil
      end

    Subject.next(hvac.deploy, p)
  end)
  ~> snk
end

defdag stream_temp(src, snk) do
  src
  ~> map(fn _ -> Thermostat.read_temperature() end)
  ~> snk
end

defdag stream_temp_temp_to_hvac(src, snk, temp_sink) do
  src
  ~> filter(fn {event, device} ->
    self = Potato.Network.Meta.get_local_nd()
    event == :join and device.room == self.room and device.type == :thermostat
  end)
  ~> map(fn {:join, thermometer} ->
    p =
      program do
        deploy(stream_temp, src: Creek.Source.range(1, :inifinity, 0, 1000), snk: temp_sink)
      end

    Subject.next(thermometer.deploy, p)
  end)
  ~> snk
end

defdag print_temperature(src, snk) do
  src
  ~> average()
  ~> debug
  ~> snk
end

###############################################################################
# Set ventilation to 20%

defdag update_ventilation(src, snk, desired) do
  src
  ~> filter(fn {event, device} ->
    self = Potato.Network.Meta.get_local_nd()
    event == :join and device.room == self.room and device.type == :ventilation
  end)
  ~> map(fn {:join, ventilation} ->
    p =
      program do
        Ventilation.set_ventilation(desired)
      end

    Subject.next(ventilation.deploy, p)
  end)
  ~> snk
end
```

```
############################################################################
# Turn off the room computer

defdag shutdown_computer(src, snk) do
  src
  ~> filter(fn {event, device} ->
    self = Potato.Network.Meta.get_local_nd()
    event == :join and device.room == self.room and device.type == :computer
  end)
  ~> map(fn {:join, computer} ->
    IO.puts("Deploying on computer #{inspect(self())}")

    p =
      program do
        Computer.turn_off()
      end

    Subject.next(computer.deploy, p)
  end)
  ~> snk
end

############################################################################
# Set the beamer to USB C.

defdag set_beamer(src, snk, source) do
  src
  ~> filter(fn {event, device} ->
    self = Potato.Network.Meta.get_local_nd()
    event == :join and device.room == self.room and device.type == :beamer
  end)
  ~> map(fn {:join, beamer} ->
    p =
      program do
        Beamer.set_output(source)
      end

    Subject.next(beamer.deploy, p)
  end)
  ~> snk
end

############################################################################
# Dim lights in rows.
defdag dim_lights(src, snk, rows, brightness) do
  src
  ~> filter(fn {event, device} ->
    self = Potato.Network.Meta.get_local_nd()
    event == :join and device.room == self.room and device.type == :light and device.row in rows
  end)
  ~> map(fn {:join, light} ->
    p =
      program do
        LightBulb.brightness(brightness)
      end

    Subject.next(light.deploy, p)
  end)
  ~> snk
end

############################################################################
# Disable internet for students.

defdag disable_wifi(src, snk, allowed, disallowed) do
  src
```

```
    ~> filter(fn {event, device} ->
      self = Potato.Network.Meta.get_local_nd()
      event == :join and device.room == self.room and device.type == :accesspoint
    end)
    ~> map(fn {:join, accesspoint} ->
      p =
        program do
          AccessPoint.filter(allowed, disallowed)
        end

      Subject.next(accesspoint.deploy, p)
    end)
    ~> snk
  end

  ##############################################################################
  # Run

  def run() do
    init()

    # When Remus enters, the temperature is set to 20 degrees.
    kill_switch = Creek.Source.gatherer()
    temps = Creek.Source.gatherer()
    deploy(update_hvac, src: Net.network(), snk: kill_switch, temp_source: temps, desired: 20.0)
    deploy(stream_temp_temp_to_hvac, src: Net.network(), snk: kill_switch, temp_sink: temps)

    # Turn off all the plugs.
    deploy(disable_plugs, src: Net.network(), snk: Creek.Sink.ignore(nil))

    # Deploy the hvac stream.

    # Print the temperatures locally for debugging.
    deploy(print_temperature, src: temps, snk: kill_switch)

    # Set the ventilation to 20%.
    # deploy(update_ventilation, src: Net.network(), snk: Creek.Sink.ignore(nil), desired: 20.0)

    # Turn off computer in room, but only do it once.
    deploy(shutdown_computer, src: Net.network(), snk: Creek.Sink.first(nil))

    # Set the beamer to USB C, but only do it once.
    deploy(set_beamer, src: Net.network(), snk: Creek.Sink.first(nil), source: :usb_c)

    # Dim the first three rows of lights.
    deploy(dim_lights, src: Net.network(), snk: Creek.Sink.first(nil), rows: [1, 2, 3])

    # Disable WiFi
    deploy(disable_wifi, src: Net.network(), snk: Creek.Sink.first(nil), allowed: [:teachers], disallowed: [:students
    nil
  end

  # Simlutes the phone moving around in the building.
  def location_simulator() do
    spawn(fn ->
      for i <- 1..10000 do
        IO.puts("Changing room to room_#{i}")
        Potato.Network.Meta.set_local_nd_field(:room, "room_#{i}")
        Process.sleep(10000)
      end
    end)
  end
end
```

```
defmodule LightBulb do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  require Logger
  use Creek
  use Potato.DSL

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :philips,
      type: :light,
      row: Keyword.get(opts, :row),
      name: "light in room #{Keyword.get(opts, :room)}",
      room: Keyword.get(opts, :room),
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run(room, row) do
    init(room: room, row: row)
  end

  def brightness(brightness) do
    IO.puts("Setting brightness to #{brightness}%")
  end
end
```

```
defmodule Thermostat do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  require Logger
  use Creek
  use Potato.DSL

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :google,
      type: :thermostat,
      name: "thermostat",
      room: Keyword.get(opts, :room),
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run(room) do
    init(room: room)
  end

  def read_temperature() do
    IO.puts("Reading")
    :rand.uniform_real() * 30
  end
end
```

```
defmodule Plug do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  require Logger
  use Creek
  use Potato.DSL

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :ikea,
      type: :plug,
      name: "smart plug",
      room: Keyword.get(opts, :room),
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run(room) do
    init(room: room)
  end

  def turn_on() do
    IO.puts("Plug turning on")
  end

  def turn_off() do
    IO.puts("Plug turning off")
  end
end
```

```
defmodule Beamer do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  require Logger
  use Creek
  use Potato.DSL

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :epson,
      name: "beamer in room 3",
      type: :beamer,
      room: Keyword.get(opts, :room),
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run(room) do
    init(room: room)
  end

  def set_output(output) do
    IO.puts("Setting output to #{output}")
  end
end
```

```
defmodule AccessPoint do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  require Logger
  use Creek
  use Potato.DSL

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :unifi,
      type: :accesspoint,
      name: "access point in room #{Keyword.get(opts, :room)}",
      room: Keyword.get(opts, :room),
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run(room) do
    init(room: room)
  end

  def filter(allowed, disallowed) do
    for a <- allowed do
      IO.puts("Allowing #{a}")
    end

    for d <- disallowed do
      IO.puts("Disallowing #{d}")
    end
  end
end
```

```
defmodule Computer do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  require Logger
  use Creek
  use Potato.DSL

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :epson,
      type: :computer,
      name: "computer in room 3",
      room: Keyword.get(opts, :room),
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run(room) do
    init(room: room)
  end

  def turn_off() do
    IO.puts("Turning off computer")
  end
end
```

Scenario 2: Noisy Campus

```
defmodule MeasurementStream do
  use Creek
  use Creek.MetaBehaviour

  def is_xml?({:xml, _}), do: true
  def is_xml?(_), do: false

  def is_json?({:json, _}), do: true
  def is_json?(_), do: false

  def xml_to_float({:xml, value}), do: value
  def json_to_float({:json, value}), do: value

  dag not_next as filter(&(not match?({p, :tick}, &1)))
                  ~> base

  fragment next as filter(&match?({p, :tick}, &1))
                   ~> base()
                   ~> map(fn base_result ->
                     case base_result do
                       {p, {state, :complete}} ->
                         {p, {state, :complete}}

                       {p, {state, :tick, value}} ->
                         decoded =
                           cond do
                             is_xml?(value) -> xml_to_float(value)
                             is_json?(value) -> json_to_float(value)
                             true -> value
                           end

                         {p, {state, :tick, decoded}}
                     end
                   end)

  dag encoding_meta(
        as dup
           ~> (next ||| not_next)
           ~> merge
      )

  defdag operator(src, snk) do
    src
    ~> base
    ~> effects
    ~> snk
  end

  defdag source(src, snk) do
    src
    ~> encoding_meta
    ~> effects
    ~> snk
  end

  defdag sink(src, snk) do
    src
    ~> base
    ~> effects
    ~> snk
  end

  # end
end

defmodule Slm.Streams do
```

```
  use Creek
  execution(MeasurementStream)

  defdag stream_measurements(src, snk) do
    src
    ~> snk
  end
end


defmodule VUB do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  alias Potato.Network.Observables, as: Net
  require Logger
  use Creek
  use Potato.DSL
  alias Creek.Source.Subject, as: Subject
  alias Creek.Source, as: Source

  def init(opts \\ []) do
    # Our node descriptor.
    nd = %{
      hardware: :update_server,
      type: :update_server,
      name: "update server",
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  ################################################################################
  # Update all devices, if necessary.

  defdag update_phones(src, snk, update, version) do
    src
    ~> filter(fn {event, device} ->
      event == :join and device.type == :smartphone and device.version < version
    end)
    ~> map(fn {:join, phone} ->
      Subject.next(phone.deploy, update)
    end)
    ~> snk
  end

  ################################################################################
  # Get all windows in this room.

  defdag windows(src, snk, room, update) do
    src
    ~> filter(fn {event, device} ->
      event == :join and device.type == :window and device.room == room
    end)
    ~> map(fn {:join, window} ->
      Subject.next(window.deploy, update)
    end)
    ~> snk
  end

  ################################################################################
  # Average soundlevel on campus.

  defdag update_slms(src, snk, measurements_sink) do
    src
    ~> filter(fn {event, device} ->
      event == :join and device.type == :slm
    end)
    ~> map(fn {_event, device} -> device end)
    ~> map(fn slm ->
      p =
        program do
          # Create stream of measurements.
          measurement_stream =
            Creek.Source.function(fn ->
              SoundLevelMeter.measure()
            end)

          Process.sleep(5000)
          deploy_module(Slm.Streams, :stream_measurements, src: measurement_stream, snk: measurements_sink)
        end

      Subject.next(slm.deploy, p)
    end)
    ~> snk
  end

  ###############################################################################
  # Listen on GUI for options.

  defdag process_measurements(src, on, off) do
    src
    ~> average(10)
    ~> average(60)
    ~> map(fn x ->
      IO.inspect("Average: #{x}")
      x
    end)
    ~> dup(2)
    ~> (filter(fn m -> m > 50 end) ~> on ||| filter(fn m -> m <= 50 end) ~> off)
  end

  defdag listen_toggle_event(src, snk) do
    src
    ~> map(fn x ->
      IO.puts("GUI event: #{inspect(x)}")
      x
    end)
    ~> map(fn event ->
      # Gather the average soundlevel.
      measurements_sink = Creek.Source.gatherer()

      on =
        Creek.Sink.each(fn measurement ->
          IO.puts("Closing windows")
        end)

      off =
        Creek.Sink.each(fn measurement ->
          IO.puts("Opening windows")
          # open_windows()
        end)

      deploy(process_measurements, src: measurements_sink, on: on, off: off)
      deploy(update_slms, src: Net.network(), snk: Creek.Sink.ignore(nil), measurements_sink: measurements_sink)

      nil
    end)
    ~> snk
  end

  ###############################################################################
  # Entry Point

  def run() do
    init([])

    # Deploy an update to all devices.
    update = program do
        IO.puts("Patching GUI for additional button.")
        Potato.Network.Meta.set_local_nd_field(:version, "1.0.1")


        # Listen for he GUI event of enabling the windows.
        deploy(listen_toggle_event, src: Process.whereis(:gui_events), snk: Creek.Sink.ignore(nil))
      end

    deploy(update_phones, src: Net.network(), snk: Creek.Sink.ignore(nil), update: update, version: "1.0.1")
    nil
  end
end
```

```
defmodule SoundLeveLMeter do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  require Logger
  use Creek
  use Potato.DSL

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: Keyword.get(opts, :hardware),
      type: Keyword.get(opts, :type),
      name: "slm",
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run(brand, type) do
    init(hardware: brand, type: :slm)
  end

  # This function mocks a measurement.
  # We cannot have multiple modules in the same codebase with the same name.
  # We emulate different implementations here.
  def measure() do
    case Potato.Network.Meta.get_local_nd().hardware do
      :typea ->
        {:json, :rand.uniform_real() * 60}

      :typeb ->
        {:xml, :rand.uniform_real() * 60}
      :foo ->
        :rand.uniform_real * 60
      end
  end
end
```

```
defmodule Window do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  alias Potato.Network.Observables, as: Net
  require Logger
  use Creek
  use Potato.DSL
  alias Creek.Source.Subject, as: Subject
  alias Creek.Source, as: Source

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :window,
      type: :window,
      name: "window",
      room: Keyword.get(opts, :room),
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run(room) do
    init(room: room)
  end
end
```

```
defmodule SmartPhone do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  alias Potato.Network.Observables, as: Net
  require Logger
  use Creek
  use Potato.DSL
  alias Creek.Source.Subject, as: Subject
  alias Creek.Source, as: Source

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :android,
      type: :smartphone,
      name: "android phone of Boole",
      version: "1.0.0",
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run() do
    init([])

    # Source that emulates events from a GUI.
    gui_events = Creek.Source.subject()
    Process.register(gui_events, :gui_events)
  end

  ###############################################################################
  # DAG for average sound level.

  defdag average_soundlevel(src, snk, measurements) do
    src
    ~> filter(fn {event, device} ->
      event == :join and device.name == "slm"
    end)
    ~> map(fn {:join, slm} ->
      p =
        program do
          src =
            Creek.Source.function(fn ->
              res = SoundLevelMeter.measure()
              Process.sleep(1000)
              res
            end)

          deploy_module(Slm.Streams, :stream_measurements, src: src, snk: measurements)
        end

      Subject.next(slm.deploy, p)
      nil
    end)
    ~> snk
  end

  def enable_option() do
    # Deploy a DAG to compute the average sound level at the campus.
    m = Creek.Sink.each(fn x -> IO.inspect(x) end)
    deploy(average_soundlevel, src: Net.network(), snk: Creek.Sink.ignore(nil), measurements: m)
  end
end
```

Scenario 3: Student Tracking

```
defmodule Student do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  require Logger
  use Creek
  use Potato.DSL

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :smartphone,
      type: :smartphone,
      name: "smartphone",
      role: :student,
      room: nil,
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run() do
    init([])

    gui_events = Creek.Source.subject(name: :gui_events)
    push_notifications = Creek.Source.subject(name: :notifications)
    nil
  end

end
```

```
defmodule Laptop do
  @moduledoc """
  Code that represents a sensor in a network. Not deployed with anything but utility software.
  """
  alias Potato.Network.Observables, as: Net
  use Creek
  use Potato.DSL
  alias Creek.Source.Subject, as: Subject
  alias Creek.Source, as: Source

  def init(opts) do
    # Our node descriptor.
    nd = %{
      hardware: :laptop,
      type: :laptop,
      name: "laptop",
      room: "room_3",
      uuid: ?a..?z |> Enum.shuffle() |> Enum.take(5) |> to_string
    }

    Potato.Network.Meta.set_local_nd(nd)
  end

  def run(room) do
    init(room)
    ping_students()
    nil
  end

  ##############################################################################
  # Students that are supposed to be here, but are not here.
  defdag missing_students(src, snk, room) do
    src
    ~> filter(fn {event, device} ->
      event in [:join, :update] and
        device.type == :smartphone and
        Map.has_key?(device, :role) and device.role == :student and
        Map.has_key?(device, :next_lecture) and device.next_lecture.room == room and
        Map.has_key?(device, :room) and device.room != room
    end)
    ~> map(fn {_, device} -> device end)
    ~> snk
  end

  ##############################################################################
  # Send notification to the students from the laptop.

  defdag query_students(src, snk) do
    src ~> snk
  end

  ##############################################################################
  # Capture the reply on the smartphone.

  defdag capture_reply(src, snk) do
    src
    ~> filter(&Kernel.match?({:incoming}, &1))
    ~> map(fn _ -> {:incoming, Potato.Network.Meta.get_local_nd().uuid} end)
    ~> snk
  end

  ##############################################################################
  # Gather replies on the laptop.

  defdag gather_responses(src, timer, snk) do
    ((timer ~> debug ~> take(1)) ||| src)
    ~> zipLatest()
    ~> map(fn {_, {:incoming, uuid}} -> IO.puts("#{uuid} is coming!") end)
    ~> snk
  end

  def ping_students() do
    # Local stream of students that are supposed to be in class.
    students = Creek.Source.gatherer()
    replies = Creek.Source.gatherer()

    # Listen for replies for a minute.
    deploy(gather_responses, src: replies, timer: Creek.Source.delay(10_000), snk: Creek.Sink.ignore(nil))

    # Query the missing students that are supposed to be in class.
    querier =
      Creek.Sink.each(fn student ->
        p = program do
          Creek.Source.Subject.next(Process.whereis(:notifications), "Are you coming to class?")
          deploy(capture_reply, src: Process.whereis(:gui_events), snk: replies)
          nil
        end

        Creek.Source.Subject.next(student.deploy, p)
      end)

    deploy(query_students, src: students, snk: querier)

    # Stream the missing students.
    deploy(missing_students, src: Net.network(), snk: students, room: Potato.Network.Meta.get_local_nd().room)
  end
end
```

Bibliography

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008. doi: 10.1109/IEEESTD.2008.4610935.

[2] Reactive streams. `http://www.reactive-streams.org/`, 2014. [Online; accessed 17-February-2020].

[3] Elixir programming language. `https://hexdocs.pm/elixir/Kernel.html`, 2018. [Online; accessed 19-April-2018].

[4] Stackexchange 2020 developer survey. `https://insights.stackoverflow.com/survey/2020`, 2020. [Online; accessed 27-May-2020].

[5] S. M. Abu Adnan Abir, Adnan Anwar, Jinho Choi, and A. S. M. Kayes. Iot-enabled smart energy grid: Applications and challenges. *IEEE Access*, 9:50961–50981, 2021. doi: 10.1109/ACCESS.2021.3067331. URL `https://doi.org/10.1109/ACCESS.2021.3067331`.

[6] Erwin Aitenbichler, Jussi Kangasharju, and Max Mühlhäuser. Mundocore: A light-weight infrastructure for pervasive computing. *Pervasive and Mobile Computing*, 3(4):332–361, 2007. ISSN 1574-1192. doi: https://doi.org/10.1016/j.pmcj.2007.04.002. URL `https://www.sciencedirect.com/science/article/pii/S1574119207000296`. Middleware for Pervasive Computing.

[7] Sven Akkermans, Stefanos Peros, Nicolas J. Small, Wouter Joosen, and Danny Hughes. Supporting iot application middleware on edge and cloud infrastructures. In Nico Herzberg, Christoph Hochreiner, Oliver Kopp, and Jörg Lenhard, editors, *Proceedings of the 10th Central European Workshop on Services and their Composition, Dresden, Germany, February 8-9, 2018*,

volume 2072 of *CEUR Workshop Proceedings*, pages 40–46. CEUR-WS.org, 2018. URL http://ceur-ws.org/Vol-2072/paper7.pdf.

[8] Ian F. Akyildiz and Ismail H. Kasimoglu. Wireless sensor and actorwe refer to entities that can act on the network as actors they are sometimes referred to as actuators in related literature. networks: research challenges. *Ad Hoc Networks*, 2(4):351–367, 2004. ISSN 1570-8705. doi: https://doi.org/10.1016/j.adhoc.2004.04.003. URL https://www.sciencedirect.com/science/article/pii/S1570870504000319.

[9] Jameela Al-Jaroodi and Nader Mohamed. Middleware is STILL everywhere!!! *Concurr. Comput. Pract. Exp.*, 24(16):1919–1926, 2012. doi: 10.1002/cpe.2817. URL https://doi.org/10.1002/cpe.2817.

[10] Jameela Al-Jaroodi, Nader Mohamed, Imad Jawhar, and Sanja Lazarova-Molnar. Software engineering issues for cyber-physical systems. In *2016 IEEE International Conference on Smart Computing, SMARTCOMP 2016, St Louis, MO, USA, May 18-20, 2016*, pages 1–6. IEEE Computer Society, 2016. doi: 10.1109/SMARTCOMP.2016.7501717. URL https://doi.org/10.1109/SMARTCOMP.2016.7501717.

[11] Jamal Al Qundus, Kosai Dabbour, Shivam Gupta, Régis Meissonier, and Adrian Paschke. Wireless sensor network for ai-based flood disaster detection. *Annals of Operations Research*, pages 1–23, 2020.

[12] Mussab Alaa, AA Zaidan, BB Zaidan, Mohammed Talal, and MLM Kiah. A review of smart home applications based on internet of things. *Journal of Network and Computer Applications*, 97:48–65, 2017.

[13] Salman Ali, Saad Bin Qaisar, Husnain Saeed, Muhammad Farhan Khan, Muhammad Naeem, and Alagan Anpalagan. Network challenges for cyber physical systems with tiny wireless devices: A case study on reliable pipeline condition monitoring. *Sensors*, 15(4):7172–7205, 2015. doi: 10.3390/s150407172. URL https://doi.org/10.3390/s150407172.

[14] Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel reification: A reflective model for distributed computation. In *1998 IEEE International Performance, Computing and Communications Conference. Proceedings (Cat. No. 98CH36191)*, pages 32–36. IEEE, 1998.

[15] Giuseppe Attardi, Cinzia Bonini, Maria Rosario Boscotrecase, Tito Flagella, and Mauro Gaspari. Metalevel programming in clos. In *ECOOP*, volume 89, pages 243–256, 1989.

[16] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, 2013.

[17] Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. Streams a la carte: Extensible pipelines with object algebras. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 591–613. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.591.

[18] Borja Bordel, Ramón Alcarria, Tomás Robles, and Diego Martín. Cyber-physical systems: Extending pervasive sensing from control theory to the internet of things. *Pervasive Mob. Comput.*, 40:156–184, 2017.

[19] Kenneth A. Bowen and Tobias Weinberg. A meta-level extension of prolog. In *Proceedings of the 1985 Symposium on Logic Programming, Boston, Massachusetts, USA, July 15-18, 1985*, pages 48–53. IEEE-CS, 1985.

[20] Gilad Bracha and David M. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In John M. Vlissides and Douglas C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 331–344. ACM, 2004. doi: 10.1145/1028976.1029004.

[21] Manfred Broy and Albrecht Schmidt. Challenges in engineering cyber-physical systems. *Computer*, 47(2):70–72, 2014. doi: 10.1109/MC.2014.30.

[22] Davide Calvaresi, Paolo Sernani, Mauro Marinoni, Andrea Claudi, Alessio Balsini, Aldo Franco Dragoni, and Giorgio Buttazzo. A framework based on real-time os and multi-agents for intelligent autonomous robot competitions. In *2016 11th IEEE symposium on industrial embedded systems (SIES)*, pages 1–10. IEEE, 2016.

[23] Davide Calvaresi, Mauro Marinoni, Arnon Sturm, Michael Schumacher, and Giorgio Buttazzo. The challenge of real-time multi-agent systems for enabling iot and cps. In *Proceedings of the International Conference on Web Intelligence*, WI '17, page 356–364, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349512. doi: 10.1145/3106426.3106518. URL `https://doi.org/10.1145/3106426.3106518`.

[24] Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In Jan Vitek, editor, *Objects, Models, Components,*

Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings, volume 6141 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2010. doi: 10.1007/978-3-642-13953-6_3. URL `https://doi.org/10.1007/978-3-642-13953-6_3`.

[25] Walter Cazzola et al. Communication-oriented reflection: a way to open up the rmi mechanism. 2000.

[26] Saurabh Chauhan, Pankesh Patel, Flávia C Delicato, and Sanjay Chaudhary. A development framework for programming cyber-physical systems. In *Proceedings of the 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems*, pages 47–53. ACM, 2016.

[27] S. Chetan, J. Al-Muhtadi, R. Campbell, and M.D. Mickunas. Mobile gaia: a middleware for ad-hoc pervasive computing. In *Second IEEE Consumer Communications and Networking Conference, 2005. CCNC. 2005*, pages 223–228, 2005. doi: 10.1109/CCNC.2005.1405173.

[28] Paolo Costa, Luca Mottola, Amy L. Murphy, and Gian Pietro Picco. Teenylime: transiently shared tuple space middleware for wireless sensor networks. In Vinny Cahill and Sam Michiels, editors, *Proceedings of the First International Workshop on Middleware for Sensor Networks, MidSens 2006, November 28, 2006, Melbourne, Australia, Co-located with Middleware 2006*, volume 218 of *ACM International Conference Proceeding Series*, pages 43–48. ACM, 2006. doi: 10.1145/1176866.1176874.

[29] J.K. Cross and D.C. Schmidt. Meta-programming techniques for distributed real-time and embedded systems. In *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. (WORDS 2002)*, pages 3–10, 2002. doi: 10.1109/WORDS.2002.1000030.

[30] Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses - practical extensibility with object algebras. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2012. doi: 10.1007/978-3-642-31057-7_2.

[31] Rustem Dautov, Salvatore Distefano, Dario Bruneo, Francesco Longo, Giovanni Merlino, and Antonio Puliafito. Data processing in cyber-physical-social systems through edge computing. *IEEE Access*, 6:29822–29835, 2018. doi: 10.1109/ACCESS.2018.2839915.

[32] Adam L. Davis. *Akka Streams*, pages 57–70. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-4176-9. doi: 10.1007/978-1-4842-4176-9_6.

[33] Christophe De Troyer. Gated dag, March 2022. URL `https://doi.org/10.5281/zenodo.6360753`.

[34] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. In *European Conference on Object-Oriented Programming*, pages 230–254. Springer, 2006.

[35] Nilanjan Dey, Amira S. Ashour, Fuqian Shi, Simon James Fong, and João Manuel R. S. Tavares. Medical cyber-physical systems: A survey. *J. Medical Syst.*, 42(4):74:1–74:13, 2018. doi: 10.1007/s10916-018-0921-x. URL `https://doi.org/10.1007/s10916-018-0921-x`.

[36] Edsger W Dijkstra. On the role of scientific thought. In *Selected writings on computing: a personal perspective*, pages 60–66. Springer, 1982.

[37] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE Conference on Local Computer Networks (LCN 2004), 16-18 November 2004, Tampa, FL, USA, Proceedings*, pages 455–462. IEEE Computer Society, 2004. doi: 10.1109/LCN.2004.38.

[38] Felix Erlacher, Bernhard Weber, Jan-Thomas Fischer, and Falko Dressler. Avarange–using sensor network ranging techniques to explore the dynamics of avalanches. In *2016 12th Annual Conference on Wireless On-demand Network Systems and Services (WONS)*, pages 1–4. IEEE, 2016.

[39] Aliaa Essameldin and Khaled A. Harras. The hive: An edge-based middleware solution for resource sharing in the internet of things. In *Proceedings of the 3rd Workshop on Experiences with the Design and Implementation of Smart Objects*, SMARTOBJECTS '17, page 13–18, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351416. doi: 10.1145/3127502.3127508. URL `https://doi.org/10.1145/3127502.3127508`.

[40] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Agilla: A mobile agent middleware for self-adaptive wireless sensor networks. *ACM Trans. Auton. Adapt. Syst.*, 4(3):16:1–16:26, 2009. doi: 10.1145/1552297.1552299. URL `https://doi.org/10.1145/1552297.1552299`.

[41] John Fox. Knowledgeable machines - knowledge-based systems in artificial intelligence. randall davis douglas lenat. mcgraw-hill: 1983. pp.

490. *The Knowledge Engineering Review*, 1(2):42–44, 1984. doi: 10.1017/S0269888900000527.

[42] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In Robert S. Boyer, Edward S. Schneider, and Guy L. Steele Jr., editors, *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*, pages 348–355. ACM, 1984. doi: 10.1145/800055.802051. URL `https://doi.org/10.1145/800055.802051`.

[43] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on software engineering*, 24(5):342–361, 1998.

[44] Antonio-Javier Garcia-Sanchez, Felipe Garcia-Sanchez, Fernando Losilla, Pawel Kulakowski, Joan Garcia-Haro, Alejandro Rodríguez, José-Vicente López-Bao, and Francisco Palomares. Wireless sensor network deployment for monitoring wildlife passages. *Sensors*, 10(8):7236–7262, 2010.

[45] David Gay, Philip Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer, and David E. Culler. The nesc language: A holistic approach to networked embedded systems. In Ron Cytron and Rajiv Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 1–11. ACM, 2003. doi: 10.1145/781131.781133.

[46] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C. M. Leung. Developing iot applications in the fog: A distributed dataflow approach. In *5th International Conference on the Internet of Things, IOT 2015, Seoul, South Korea, 26-28 October, 2015*, pages 155–162. IEEE, 2015. doi: 10.1109/IOT.2015.7356560. URL `https://doi.org/10.1109/IOT.2015.7356560`.

[47] Michael Golm and Jürgen Kleinöder. Implementing real-time actors with metajava. In Jan Bosch and Stuart Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader, ECOOP'97 Workshops, Jyväskylä, Finland, June 9-13, 1997*, volume 1357 of *Lecture Notes in Computer Science*, pages 68–73. Springer, 1997. doi: 10.1007/3-540-69687-3_13. URL `https://doi.org/10.1007/3-540-69687-3_13`.

[48] Rachid Guerraoui, Benoît Garbinato, and Karim R Mazouni. Garf: A tool for programming reliable distributed applications. *IEEE concurrency*, 5(4):32–39, 1997.

[49] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. Interacting with the soa-based internet of things: Discovery,

query, selection, and on-demand provisioning of web services. *IEEE Trans. Serv. Comput.*, 3(3):223–235, 2010. doi: 10.1109/TSC.2010.3. URL `https://doi.org/10.1109/TSC.2010.3`.

[50] Levent Gürgen, Ozan Necati Günalp, Yazid Benazzouz, and Mathieu Gallissot. Self-aware cyber-physical systems and applications in smart buildings and cities. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 1149–1154. EDA Consortium San Jose, CA, USA / ACM DL, 2013. doi: 10.7873/DATE.2013.240. URL `https://doi.org/10.7873/DATE.2013.240`.

[51] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Computer Languages, Systems and Structures*, 32(2):109–124, 2006. ISSN 1477-8424. doi: https://doi.org/10.1016/j.cl.2005.10.003. URL `https://www.sciencedirect.com/science/article/pii/S1477842405000448`. Smalltalk.

[52] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973.

[53] Md. Abdus Samad Kamal, Chee Pin Tan, Tomohisa Hayakawa, Shun-Ichi Azuma, and Jun-ichi Imura. Control of vehicular traffic at an intersection using a cyber-physical multiagent framework. *IEEE Trans. Ind. Informatics*, 17(9):6230–6240, 2021. doi: 10.1109/TII.2021.3051961. URL `https://doi.org/10.1109/TII.2021.3051961`.

[54] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, DYLA '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320412. doi: 10.1145/2489798.2489802. URL `https://doi.org/10.1145/2489798.2489802`.

[55] Sungjoo Kang, Ingeol Chun, and Wontae Kim. Dynamic software updating for cyber-physical systems. In *The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*, pages 1–3, 2014. doi: 10.1109/ISCE.2014.6884473.

[56] Woochul Kang, Krasimira Kapitanova, and Sang Hyuk Son. Rdds: A real-time data distribution service for cyber-physical systems. *IEEE Trans-*

actions on Industrial Informatics, 8(2):393–405, 2012. doi: 10.1109/TII.2012.2183878.

[57] Shweta Khare, Kyoungho An, Aniruddha Gokhale, Sumant Tambe, and Ashish Meena. Reactive stream processing for data-centric publish/subscribe. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 234–245. ACM, 2015.

[58] Shweta Khare, Kyoungho An, Aniruddha S. Gokhale, Sumant Tambe, and Ashish Meena. Reactive stream processing for data-centric publish/subscribe. In Frank Eliassen and Roman Vitenberg, editors, *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, Oslo, Norway, June 29 - July 3, 2015*, pages 234–245. ACM, 2015. doi: 10.1145/2675743.2771880. URL `https://doi.org/10.1145/2675743.2771880`.

[59] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT press, 1991. ISBN 9780262610742.

[60] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyvaskyla, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997. doi: 10.1007/BFb0053381.

[61] Alexander Kirsanov, Iakov Kirilenko, and Kirill Melentyev. Robotics reactive programming with f# mono. In *Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia*, CEE-SECR '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328890. doi: 10.1145/2687233.2687249.

[62] Igor Kopestenski and Peter Van Roy. Achlys: Towards a framework for distributed storage and generic computing applications for wireless iot edge networks with lasp on grisp. In *IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2019, Kyoto, Japan, March 11-15, 2019*, pages 875–881. IEEE, 2019. doi: 10.1109/PERCOMW.2019.8730773. URL `https://doi.org/10.1109/PERCOMW.2019.8730773`.

[63] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: a taxonomy of actor models and their key properties. In Sylvan Clebsch, Travis Desell, Philipp Haller, and Alessandro Ricci,

editors, *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016*, pages 31–40. ACM, 2016. doi: 10.1145/3001886.3001890.

[64] YoungMin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: an actor platform for wireless sensor networks. In Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, editors, *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006*, pages 1297–1300. ACM, 2006. doi: 10.1145/1160633.1160871. URL `https://doi.org/10.1145/1160633.1160871`.

[65] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason L. Hill, Matt Welsh, Eric A. Brewer, and David E. Culler. Tinyos: An operating system for sensor networks. In Werner Weber, Jan M. Rabaey, and Emile H. L. Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer, 2005. doi: 10.1007/3-540-27139-2_7.

[66] Inc. Lightbend. Akka streams. `https://doc.akka.io/docs/akka/current/stream/index.html`, 2020. [Online; accessed 19-April-2020].

[67] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005. doi: 10.1145/1061318.1061322.

[68] Pattie Maes. Concepts and experiments in computational reflection. In Norman K. Meyrowitz, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), Orlando, Florida, USA, October 4-8, 1987, Proceedings*, pages 147–155. ACM, 1987. doi: 10.1145/38765.38821. URL `https://doi.org/10.1145/38765.38821`.

[69] Andrea Maglie. Rxjava and android. In *Reactive Java Programming*, pages 95–105. Apress, Berkeley, CA, 2016. ISBN 978-1-4842-1428-2. doi: 10.1007/978-1-4842-1428-2_6.

[70] Jeff McAffer. Meta-level programming with coda. In Walter G. Olthoff, editor, *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings*, volume 952 of *Lecture Notes in Computer Science*, pages 190–214. Springer, 1995. doi: 10.1007/3-540-49538-X_10. URL `https://doi.org/10.1007/3-540-49538-X_10`.

[71] Samir Medjiah and Christophe Chassot. On the enhancement of non-functional requirements for cloud-assisted middleware-based iot and other applications. *CoRR*, abs/2010.16147, 2020. URL `https://arxiv.org/abs/2010.16147`.

[72] Nader Mohamed, Jameela Al-Jaroodi, Sanja Lazarova-Molnar, and Imad Jawhar. Middleware challenges for cyber-physical systems. *Scalable Comput. Pract. Exp.*, 18(4):331–346, 2017.

[73] Roberto Morabito, Riccardo Petrolo, Valeria Loscrì, and Nathalie Mitton. Reprint of : Legiot: A lightweight edge gateway for the internet of things. *Future Gener. Comput. Syst.*, 92:1157–1171, 2019. doi: 10.1016/j.future.2018.10.020. URL `https://doi.org/10.1016/j.future.2018.10.020`.

[74] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3):19:1–19:51, 2011. doi: 10.1145/1922649.1922656.

[75] Angeles G. Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical modeling of pipeline parallelism. In *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA*, pages 281–290. IEEE Computer Society, 2009. doi: 10.1109/PACT.2009.28.

[76] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 489–498. IEEE, 2007.

[77] Andreas Pamboris, Charalampos Kozis, and Herodotos Herodotou. Cuttlefish: A flexible and lightweight middleware for combining heterogeneous iot devices. In *CCNC*, pages 1–6. IEEE, 2020.

[78] Gerardo Pardo-Castellote. OMG data-distribution service: Architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops (ICDCS 2003 Workshops), 19-22 May 2003, Providence, RI, USA*, pages 200–206. IEEE Computer Society, 2003. doi: 10.1109/ICDCSW.2003.1203555. URL `https://doi.org/10.1109/ICDCSW.2003.1203555`.

[79] Pankesh Patel and Damien Cassou. Enabling high-level application development for the internet of things. *Journal of Systems and Software*, 103:62–84, 2015. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2015.01.027. URL `https://www.sciencedirect.com/science/article/pii/S0164121215000187`.

[80] Per Persson and Ola Angelsmark. Calvin – merging cloud and iot. *Procedia Computer Science*, 52:210–217, 2015. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2015.05.059. URL https://www.sciencedirect.com/science/article/pii/S1877050915008595. The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015).

[81] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Clarke. Middleware for internet of things: A survey. *IEEE Internet Things J.*, 3(1):70–95, 2016. doi: 10.1109/JIOT.2015.2498900. URL https://doi.org/10.1109/JIOT.2015.2498900.

[82] ReactiveX. Reactivex: Observables done right. https://reactivex.io/, 2014. [Online; accessed 19-April-2020].

[83] ReactiveX. Rxjava. https://github.com/ReactiveX/RxJava, 2020. [Online; accessed 19-April-2020].

[84] IBM Emerging Technology Services. Node-red. https://nodered.org/, 2018. [Online; accessed 19-April-2018].

[85] Yuanchun Shi, Weikai Xie, Guangyou Xu, Runting Shi, Enyi Chen, Yanhua Mao, and Fang Liu. The smart classroom: Merging technologies for seamless tele-education. *IEEE Pervasive Comput.*, 2(2):47–55, 2003. doi: 10.1109/MPRV.2003.1203753.

[86] Thirunavukkarasu Sivaharan, Gordon S. Blair, and Geoff Coulson. GREEN: A configurable and re-configurable publish-subscribe middleware for pervasive computing. In *OTM Conferences (1)*, volume 3760 of *Lecture Notes in Computer Science*, pages 732–749. Springer, 2005.

[87] Alessandro Sivieri, Luca Mottola, and Gianpaolo Cugola. Building internet of things software with eliot. *Computer Communications*, 89:141–153, 2016.

[88] Brian Cantwell Smith. Reflection and semantics in lisp. In Ken Kennedy, Mary S. Van Deusen, and Larry Landweber, editors, *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984*, pages 23–35. ACM Press, 1984. doi: 10.1145/800017.800513. URL https://doi.org/10.1145/800017.800513.

[89] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35, 1984.

[90] Daniel Charles Sturman. *Modular specification of interaction policies in distributed computing*. University of Illinois at Urbana-Champaign, 1996.

[91] Thiago Teixeira, Sara Hachem, Valérie Issarny, and Nikolaos Georgantas. Service oriented middleware for the internet of things: A perspective. In Witold Abramowicz, Ignacio M. Llorente, Mike Surridge, Andrea Zisman, and Julien Vayssière, editors, *Towards a Service-Based Internet*, pages 220–229, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24755-2.

[92] Christophe De Troyer, Jens Nicolay, and Wolfgang De Meuter. First-class reactive programs for CPS. In Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, and Lukasz Ziarek, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Vancouver, BC, Canada, October 23, 2017*, pages 21–26. ACM, 2017. doi: 10.1145/3141858.3141862.

[93] Christophe De Troyer, Jens Nicolay, Wolfgang De Meuter, and Christophe Scholliers. Abstractions for distributed event-driven applications: Position paper. In Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter, editors, *Companion to the first International Conference on the Art, Science and Engineering of Programming, Programming 2017, Brussels, Belgium, April 3-6, 2017*, pages 18:1–18:2. ACM, 2017. doi: 10.1145/3079368.3079395.

[94] Christophe De Troyer, Jens Nicolay, and Wolfgang De Meuter. Building iot systems using distributed first-class reactive programming. In *2018 IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2018, Nicosia, Cyprus, December 10-13, 2018*, pages 185–192. IEEE Computer Society, 2018. doi: 10.1109/CloudCom2018.2018.00045.

[95] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time frp. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, page 146–156, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581134150. doi: 10.1145/507635.507654. URL `https://doi.org/10.1145/507635.507654`.

[96] Kamin Whitehouse, Feng Zhao, and Jie Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In Kay Römer, Holger Karl, and Friedemann Mattern, editors, *Wireless Sensor Networks*, pages 5–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32159-0.

[97] Roel Wuyts and Stéphane Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *In ECOOP 2001 Inter-*

national workshop on MultiParadigm Programming with Object-Oriented Languages, pages 81–96, 2001.

[98] Yasuhiko Yokote. The apertos reflective operating system: The concept and its implementation. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '92, page 414–434, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0201533723. doi: 10.1145/141936.141970. URL `https://doi.org/10.1145/141936.141970`.

[99] Tao Zheng, Jian Wan, Jilin Zhang, Congfeng Jiang, and Gangyong Jia. A survey of computation offloading in edge computing. In *2020 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pages 1–6, 2020. doi: 10.1109/CITS49457.2020.9232457.