University of Zurich[UZH]

A Domain Specific Language for the Development of Interdependent Human Computation Processes

Thesis November 11, 2014

Marc Tobler
of Glattfelden ZH, Switzerland

Student-ID: 06-910-764
marc_tobler@bluewin.ch

Advisor: **Patrick de Boer**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

First and foremost I want to thank my tutor Patrick de Boer for his continuous feedback and support all the way from the early concepts of the domain specific language to the final draft of this thesis. Without him, it would not have been possible. I also want to thank Professor Abraham Bernstein, head of the Dynamic and Distributed Information Systems Group (DDIS) at the University of Zurich, for providing me with the chance to write this thesis. I have always enjoyed his lectures and I am happy to now conclude my academic career with a project in his group. Additionally, I want to mention Patrick Minder, who has, especially in the beginning of my work, but also throughout the process, provided me with ongoing support in regards to CrowdLang. Lastly, I want to thank my family and friends for their moral support. Specifically, I want to mention Pascal Muther and Michael Keller, who - while working on their own theses - helped me keep my motivation when I needed it the most.

Zusammenfassung

Crowdsourcing Plattformen wie Amazons "Mechanical Turk" oder "CrowdFlower" bieten Firmen neue Möglichkeiten zur Auslagerung von Arbeit. Diese Dienstleistungsanbieter sind sehr gut darin eine grosse Menge von einfachen, repetitiven und voneinander unabhängigen Aufgaben parallel ausführen zu lassen. Ihnen mangelt es aber an Koordinationsmechanismen, welche es ermöglichen auch gesamte Geschäftsprozesse abzuwickeln. Die hier präsentierte Arbeit stellt eine domänenspezifische Programmiersprache vor, die es erlaubt auch komplexe Abläufe menschlicher Zusammenarbeit (human computation) zu koordinieren. Wir basieren die Sprache auf den Konzepten von *CrowdLang* und evaluieren ihre Fähigkeiten anhand von Beispielimplementationen eines Algorithmus zur Korrektur von Texten und eines Prozesses zu Kategorisierung von Bildern.

Abstract

Crowdsourcing platforms like Amazon's Mechanical Turk or CrowdFlower have provided companies with new opportunities to source their work load. But while they allow the completion of massive amounts of work in parallel, the tasks performed on said websites are mainly of simple and isolated nature. The lack of coordination mechanisms has hindered the advance of crowdsourcing into application areas with more complex and interdependent working processes. This thesis provides a domain specific language for the orchestration of complex human computation processes based of the concepts of *CrowdLang*. We present the capabilities of our language using example implementations of a proof reading algorithm as well as an image categorization application.

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Goal of this Thesis 2

1.3 Outline 3

2 Related Work **5**

3 Design Decisions **9**

3.1 Finding the Foundation 9

3.2 CrowdLang 9

3.3 Components 11

3.4 The Akka Framework 11

3.4.1 Actors 11

3.5 Sequential Approach 12

3.6 Issues with the Sequential Approach 12

3.6.1 Non-local Dependencies and Information Propagation 12

3.6.2 Dead-Ends 13

3.7 Hierarchical Approach 14

3.8 Mapping Sequential Patterns into Hierarchical Patterns 14

3.9 Summary 17

4 Implementation **19**

4.1 Akka in the Component Trait 19

4.2 Operators 22

4.3 Tasks 24

4.4 Crowd Computing Platform Integration 27

4.5 Event Queue 29

5 Evaluation **31**

5.1 Implemented Algorithms 31

5.2 CrowdFlower Integration 31

5.3 Find Fix Verify 32

5.3.1 Concept 32

5.3.2 Experimental Setup 34
5.3.3 Results 36
5.4 Image Labelling 38
5.4.1 Concept 38
5.4.2 Experimental Setup 38
5.4.3 Results 39
5.5 Discussion 39

6 Future Work **41**

7 Conclusions **43**

1

Introduction

1.1 Motivation

Automation has been one of the most important driving factors for the ever increasing productivity of our economy. What started with constructions like windmills and steam engines replacing or changing the need for physical labour has over the last century more and more started to influence mental labour as well. Computers have changed the requirements for white collar jobs in drastic ways, even making some professions obsolete. Even the eponymous occupation of "computers" - people who were hired to manually carry out computation - has disappeared for the most part.

With constantly increasing computing power and new developments in artificial intelligence it is to be expected that this process will continue replacing knowledge work with computers in the future. But there remains the ever changing set of problems that humans can still be better at than computers. Repetitive "simple" tasks that computers are not capable of performing reliably enough - yet. Nevertheless, companies in competitive environments are under the constant pressure of increasing productivity and efficiency in these areas. A variety of approaches have been pursued, including new sourcing strategies like outsourcing as well as the constant strive for division of labour to reduce the complexity and cost of each of those tasks.

Crowd sourcing could be interpreted as a combination of those two approaches. On the one hand, organizations try to decompose their tasks so that they can be performed by lower skilled workers in better quality, on the other hand they try to delegate the management and recruitment of those workers to service companies, potentially even located abroad. Crowd sourcing drives these tendencies to their extreme. The tasks are divided into ever so small chunks of work that can usually be performed by any individual in the manner of seconds or minutes at most, even without prior knowledge of the task. And the workers are almost anonymous entities from the viewpoint of the company. In their 2013 paper, Kittur et al. [Kittur et al., 2013] point out the "remarkable opportunities for improving productivity" but they also "foresee a serious risk that crowd work will fall into an intellectual framing focused on low-cost results and exploitative labor". But the crowd computing platforms - mainly websites - that were developed to provide the tools for employers to crowd source their work actually cater to the needs of workers pretty well. Workers can work from the comfort of their

home, requiring only an internet connection and an account on the websites. This opens up possibilities for many people that are required to spend much of their time at home such as stay-at-home mothers or fathers, or people who have to care for a sick family member. The payment options are also very worker friendly as any money earned can usually be paid out instantaneously. The search for work has been simplified, there is no need for lengthy application processes. Users can also work at any time of the day, allowing them to work worldwide and according to their own schedule.

The benefits of crowd computing sound promising and in fact over the past years an ever growing number of websites have been established. Probably the most well-known one is Mechanical Turk - a service that was launched by Amazon in 2005. But there are also others who have their own crowds of workers like ClickWorker, while other service providers build on top of Mechanical Turk by providing higher level functionality to employers such as selection of worker groups (e.g. high quality workers, workers from a certain region) or improved worker feedback and many more. CrowdFlower, the website which we will be using in our evaluation section, is one of these providers.

A lot of the previously mentioned platforms have been successful with many customers. But while they are perfectly capable of handling "atomic" tasks in masses, they lack functionality to enable collaboration between workers and ways of combining atomic jobs into more complex processes and algorithms. Novel approaches for modelling and programming human intelligence tasks are required to provide customers with such functionality.

There have been many attempts to develop models in the crowd research community. Frameworks like Automan [Barowy et al., 2012] and Jabberwocky [Ahmad et al., 2011] have shown interesting concepts of domain specific programming languages suited to face the challenges presented. There have also been a series of papers that focussed on specific interaction patterns between atomic tasks. CrowdForge [Kittur et al., 2011] has shown how many of these interactions can be interpreted using the well-known MapReduce pattern [Dean and Ghemawat, 2008], while the authors of TurKit [Little et al., 2009] have highlighted the power of an iterative improvement algorithm.

At the Dynamic and Distributed Information Systems Group (DDIS) at the University of Zurich, Minder, Bernstein et al. have developed the concept of the human computation language CrowdLang. CrowdLang attemps to support not only asynchronous parallelization but also the management of arbitrary dependencies among tasks and workers [Minder and Bernstein, 2012a]. Its capabilities have already been demonstrated in a series of papers, but while the concept has been refined over time, the actual implementation of the language is still a working prototype. In this thesis we strive to develop a flexible and open-sourced implementation of a domain specific language for human intelligence tasks based on the concepts of CrowdLang.

1.2 Goal of this Thesis

The goal of this thesis is to develop a domain specific language (DSL). The language will aspire to seamlessly integrate human computation performed on crowd sourcing websites

with machine computation into composite computation processes using the conceptual framework of CrowdLang [Minder and Bernstein, 2012a]. The language should help to efficiently program novel computation systems, while coping with the challenges of human work such as latency and varying quality of worker contributions. It will be evaluated using two algorithms introduced in other research papers to show its flexibility and applicability. The implemented algorithms will be an adaptation of the Find-Fix-Verify pattern introduced in the Soylent paper by Berstein et al. [Bernstein et al., 2010] and a image categorization task with confidence levels as described by Barowy et al. in their Automan platform [Barowy et al., 2012].

1.3 Outline

The remainder of the thesis will be structured as follows. Firstly, we will take a look at other research that is closely related to ours. In particular, we will show a series of languages and frameworks that have been proposed for crowd computing, such as Automan [Barowy et al., 2012], CrowdForge [Kittur et al., 2011], TurKit [Little et al., 2009] and Jabberwocky [Ahmad et al., 2011]. We will introduce the basic concepts of these frameworks and how they are important for our own approach. Additionally, we will look into papers that suggested specific patterns of interaction among human intelligence tasks, for example Soylent [Bernstein et al., 2010] or PlateMate [Noronha et al., 2011]. We will then go on to show how CrowdLang is embedded among those frameworks and try to find a solid foundation for our DSL. Thereafter, we will develop our own adapted model of CrowdLang and show how we implemented it using code examples. After we have established the core capabilities of our language we will show how it can be used to implement real applications using the CrowdFlower platform in our evaluation section. Lastly we will summarize our results and point out possibilities for future work.

2

Related Work

This chapter contains information about general research in crowd computing, frameworks and languages studied during the conception this thesis. We will discuss the concepts of said frameworks and show how they influence the design of our own.

As a starting point we should look into frameworks that try to categorize existing crowd computing solutions. Malone et. al [Malone et al., 2010] present their concept based on four overarching questions:

- *What* is being done?
- *Who* is doing it?
- *Why* are they doing it?
- *How* is it being done?

Each of those questions represent a dimension of customization along which the existing crowd computing algorithms and platforms can be placed. In all four dimensions the authors define an answer to the overarching question as a "gene". They identify a set of established genes for each dimension. The notion of genes has been applied in the CrowdLang papers, where some of them, mainly the "create" and "decide" genes have been conceptualized into patterns. As our domain specific language will build on top of the CrowdLang framework, we will also be able to support these genes.

An alternative framework has been presented by Quinn & Pedersen in their "Taxonomy of Distributed Human Computation" [Quinn and Bederson, 2009]. Similar to the overarching questions in Malone the authors here describe a set of dimensions in which existing crowd work solutions can be distinguished. Their dimensions are

- *Motivation* - which describes the different types of compensations for workers
- *Quality* of the work done
- *Aggregation* of atomic results into solutions to the global problem
- *Human Skill* of the workers involved
- *Participation Time* - how much time a worker has to invest for a typical task

- *Cognitive Load* - how much mental work load a task contains

The authors argue that each of these dimensions offer multiple choices and they characterize known approaches like *Games with a Purpose* along those dimensions. This paper provides a nice environment to describe the focus of our work. While we will not directly be concerned with the *Motivation* dimension, we will try to leave the form of compensation up to interpretation for the developers of crowd computing platform integrations (see section 4.4). For the sample integration with CrowdFlower the compensation will be per task cash payments. The *Quality* dimension will be relevant for both example algorithms that we will implement in our evaluation section. Both Find-Fix-Verify and the confidence level based image categorization from Automan try to combat the generally rather poor results generated by crowd workers, although be it with very different approaches. The main focus of our paper however will of course be the *Aggregation* dimension. While Quinn & Bedersen list some types of aggregation, the amount of possible patterns is far more extensive and the idea of CrowdLang as well as our implementation of it is to enable programmers to implement those patterns. We will leave all other three dimensions mentioned open to customization to any programmer that uses our work, as they vary strong between applications and should not be limited by a language.

Now that we were able to place our thesis into the picture of the existing human intelligence research environment, we should look at projects that pursued similar approaches to ours. The first paper we want to look at in this context is the CrowdForge paper by Kittur et al. [Kittur et al., 2011]. They developed a web-based toolkit that is built on three task primitives called partition, map and reduce. As the names already suggest, their concept is heavily inspired by MapReduce [Dean and Ghemawat, 2008]. In their framework a *partition* task will divide a larger task into smaller subtasks, a *map* task will compute results to these subtasks and the *reduce* task will merge the results into a single output. The approach models a very common interaction pattern between tasks and we will be implementing an operator called *ForEach* that is actually very similar to the concept developed in the CrowdForge paper. Relevant for our work are also the limitations they mentioned for their framework, namely iteration in the process as well as dependency between multiple subtasks. We will be addressing both those concerns with our approach by introducing the *Iterate* operator and by moving from a sequential concept to a hierarchical concept of coordination.

Another attempt at trying to model human intelligence processes is presented in the Jabberwocky framework [Ahmad et al., 2011]. It consists of three components. *Dormouse* is an interface that interacts with different crowd computing plattforms such as Amazon's Mechanical Turk and abstracts their usage for the other components. ManReduce is the intermediate layer that works similar to CrowdForge. It allows programmers to construct coordination patterns for the tasks created with Dormouse in the MapReduce scheme. The third component is called Dog, which is a top level DSL whose code compiles into ManReduce scripts and whose goal it is to make the framework easier to use and accessible to a broader user base. It works with four high-level primitives called PEOPLE, ASK, FIND, COMPUTE and scripts written in Dog resemble SQL State-

ments in their appearance. We are very impressed with the Jabberwocky framework and will use it as inspiration for much of our own design. We will create a separate module that handles the integration of crowd computing platforms just like Dormouse does, and our eventual language will solve similar tasks as ManReduce. But as we already mentioned, we will not only allow for MapReduce types of task decomposition but also provide additional coordination operators.

TurKit is a Javascript toolkit [Little et al., 2009] that allows its users easy deployment of tasks to Mechanical Turk like Dormouse does in the Jabberwocky framework. Additionally, it supports the crash and rerun model via a JavaScript database using memoization of results with the so called "once" function. It is built on the concept of iterative improvement that can be used for a variety of applications including handwriting recognition, image description, copy editing or brainstorming. While iterative improvement is also mentioned in other papers, we believe TurKit shows the best realization of the concept. As previously mentioned, we will also allow for iteration in our domain specific language using the *Iterate* operators, but as the implementation will be rather basic, future work in this area should be considered using TurKit as guidance.

While the above frameworks tried to model a wide variety of applications with their MapReduce or iteration based approaches, we should also study some papers that focus on more specific topics. The PlateMate [Noronha et al., 2011] paper describes an application that allows users to upload pictures of their meals for nutritional analysis. The system works in three steps. First, workers tag individual items in the meal by drawing a box over them. Then, for each of those tagged items, other worker identify the type of food using a given library of ingredients and meals. And finally, other workers measure the amount of food by either using weight or predefined units such as "wings" or "nuggets". The PlateMate process is another example of a complex task that we will be able to build using our DSL. The most relevant take away point for us in this paper is the hierarchical structure of tasks that they use. Their process has a top-level task which then controls the tag, identify and measure tasks, which in turn control their own subtasks. As we will see later, this kind of tree structure can be very beneficial and we will in fact compose the tasks in our DSL in a hierarchical manner using Akka actors.

Liem, Zhang and Chen [Liem et al., 2011] present an approach to audio transcription where the same bit of audio is fed into two separate paths of iterative improvement for transcription. Each contributor in one path gets the text from his predecessor and is rated by comparing his results with those of the other path. This is an interesting approach as it deals with the problem of motivation for crowd workers. The authors show that the best strategy for any worker in this environment is to give his best effort of actually solving the task. By using this technique they reach a much higher performance. The dual pathway structure is a very interesting use case as it includes non-sequential dependencies between tasks. Each contributors' rating is dependent on the results of the other path's contributors, which are not sequentially related to it. This is a core problem of a purely sequential approach which we will be discussing later in our design decisions.

Another source for a complex human computation process is introduced in Automan [Barowy et al., 2012]. Automan is an automatic crowd programming system that in-

tegrates human computation tasks as ordinary function calls into Scala. It currently supports multiple-choice and restricted free-text tasks on Mechanical Turk and automatically takes care of acceptance and rejection of worker contributions. The main concept of Automan is to continuously schedule new copies of a task on Mechanical Turk until a certain confidence level for the question at hand is reached. We believe this is a novel and promising algorithm that should be supported by any language that wishes to automate human computation processes and we will therefore use it as one of our proof of concepts in the evaluation section, where we will also go into a little more detail about the specifics of the algorithm.

On a similiar note we also decided to choose the Find-Fix-Verify pattern introduced in the Soylent paper [Bernstein et al., 2010] as our second use case. The paper itself describes the Soylent toolkit for Microsoft Word. It consists of three components named *Shortn*, *CrowdProof* and *The Human Macro*. Shortn lets the user shorten a paragraph by sending it to the crowd, whereas CrowdProof uses workers on the crowd to proofread text and The Human Macro allows offloading arbitrary word processing tasks. But as mentioned, the contribution that is relevant to our work is the Find-Fix-Verify pattern that is used in Shortn as well as CrowdProof. In this pattern the problem statement is divided into three stages with three different groups of workers assigned to each stage. In the Find stage, users are asked to identify parts of the problem statement that require work. In the case of Shortn these would be sentences that require shortening, in crowd proof that would be sentences containing errors that need correction. In the Fix stage other workers are required to correct or shorten said sentences and in the Verify stage a last group of workers have to choose the best option among a set of possible fixes provided by the previous group. We will discuss this pattern in more detail in our evaluation section.

Lastly we also want to mention that CrowdLang [Minder and Bernstein, 2012c] and [Minder and Bernstein, 2012b] is obviously closely related and of high importance to this paper. It will therefore be mentioned at multiple points in the thesis and the core concepts will be explained in our design chapter.

3

Design Decisions

The following chapter describes the theoretical concepts of CrowdLang and how we evolved these concepts into our own implementation of a domain specific language. The idea of describing a human computation process in the terms of data flows, operators and tasks is laid out and same examples of typical crowd computation algorithms are presented.

3.1 Finding the Foundation

As we pointed out in our introduction, "atomic" human intelligence tasks can be handled very well by existing crowd computing solutions. When looking at more complex human computation processes however, one can find that much of the complexity stems from the coordination between individual tasks, rather than the tasks themselves. In fact, most of the papers described in our related work have tried to find ways in which to describe the interaction patterns and coordination mechanics between individual human intelligence tasks. Some authors have tried to map the interactions used in HIT processes to known patterns from other research areas. MapReduce has been used in both CrowdForge [Kittur et al., 2011] as well as Jabberwocky [Ahmad et al., 2011], whereas TurKit [Little et al., 2009] relied on iterative improvement.

CrowdLang differentiates itself from these papers, in that it not only includes one pattern, but collects the results of previously mentioned papers and provides a total of six different operators which can be arbitrarily combined into new as well as existing interaction patterns.

3.2 CrowdLang

The core concept of CrowdLang is that each human computation process can be described in a form similar to other process languages. Processes can be described in the combination of *Tasks* and *Operators*. Tasks solve a given problem statement and represent a single unit of computation, be it machine or human computation. Operators on the other hand control the data and sequence flow between multiple tasks.

[Minder and Bernstein, 2011] Together these elements can be combined into arbitrarily large and complex processes which can solve a wide variety of problem statements.

In their papers, Minder et al. present a total of six operators: *Divide & Conquer, Aggregate, Multiply, Reduce, XOR* and *AND*. We will shortly summarize the function of each of those operators.

Divide & Conquer The Divide & Conquer operator divides a given problem statement into multiple smaller subproblems and distributes them into multiple paths to be solved in parallel.

Aggregate The Aggregate operator then collects the results of subproblems and aggregates them to a solution of the original problem statement. Together Divide & Conquer and Aggregate model a typical case of division of labour.

Multiply The Multiply operator is similar to the Divide & Conquer in that it distributes work to multiple paths of execution which can be computed in parallel. The difference between the two is that multiply, rather than decomposing the original problem into subproblems, sends a copy of the entire problem statement to each path, so that multiple solutions for the problem can be found.

Reduce The Reduce operator is then the counterpart to the Multiply operator, as it collects the various results and merges them into one. How this merging process is done depends on the exact implementation. It could potentially follow a selection approach, where workers (or a machine) choose the best solution provided, or it could pursue some sort of combination approach to fuse multiple solutions into one.

XOR and AND Minder et. all also provide some well-known control flow operators in XOR and AND, where "XOR is used to create or synchronize alternative paths; AND can be used to create and synchronize parallel paths." [Minder and Bernstein, 2012a]

Minder et al. use these operators in combination with a few distinct tasks to describe common human computation process patterns in their papers. But the concept is more powerful than that. Leaving the implementation of the tasks open to users of CrowdLang, the language allows for a wide variety of applications. In this thesis we provide a first openly available and reusable implementation of a domain specific language based on the concepts of CrowdLang. As we will see, our model has also evolved from the basic ideas contained in CrowdLang. The following sections will show how we used CrowdLang as a foundation and how adapted the design in response to problems we encountered during our implementation.

3.3 Components

Starting with the model of operators and tasks we set out to create an implementation of CrowdLang. While working on a class structure for our Scala project we quickly noticed that on a fundamental technical level, operators and tasks are not that different. Both tasks as well as operators are components of our "process network". They have to handle data and sequence flow and their behaviour should almost exclusively depend on the input data they receive in the network. In the interest of parallelization and eventually distributed execution of processes they should also be as loosely coupled as possible. All these factors lead us to pursue an approach where we summarized the commonalities of tasks and operators into the *Component* trait. All components should act event-driven to reduce coupling and possess a dedicated thread. Their behaviour is state-based and mainly dependent on the type of input they receive. All these capabilities are supported and simplified by the Akka programming framework and its actor-based computation model, which is why we decided to build our core functionality using Akka. In the following section we will shortly summarize the ideas behind Akka's actor model and how we translated the above concept of components into actors.

3.4 The Akka Framework

Akka is an open-source toolkit for the distribution and parallelization of computation. It is built around the actor model which has been introduced by Erlang [Armstrong, 2007]. It allows programmers to build event-based, fault-tolerant, scalable applications for the Java Virtual Machine [Raychaudhuri, 2013] and provides support for both Scala and Java. The basic building block of any Akka application are actors, hence we are going to take a look at them in the next section.

3.4.1 Actors

"Actors are objects which encapsulate state and behavior, they communicate exclusively by exchanging messages which are placed into the recipient's mailbox." [Akka, 2014]

Actors are event-driven, i.e. they react to messages they receive in their mailbox. Once they receive a message they process it and usually reply the result to the sender. Consider the following example:

```
class MyActor extends Actor {
  override def receive: Receive = {
    case s: String =>
      println(s)
  }
}
```

The *receive* is where programmers can specify the behaviour of an actor. In the above example, the actor can only handle messages of type String and it will print any String

it receives to the console.

In Akka, each actor is part of a hierarchy - the *ActorSystem*. At the top of it we find the so called guardian actor which can be created using a command that looks like this.

```
val system = ActorSystem("Addition")
```

Once the ActorSystem and its guardian actor are created, new actors can be added using *actorOf*.

```
//top level actor, directly under the guardian
val topLevelActor = system.actorOf(Props[MyActor])
//lower level actor, to be created from within its
//supervisor(parent)
val childActor = context.actorOf(Props[ChildActor])
```

Using these methods, a hierarchy of actors can be created, where each parent controls the work of its children. The parents take a supervisory role and can not only send work to their child actors but also control their shut-down and restart in case of an error.

3.5 Sequential Approach

Originating from the basic Akka actor model, we started implementing the interface for our operators and tasks, thereby developing the previously mentioned *Component* class. A component is an element in a process network that expects a certain type of input and then reacts according the the input given in a stateful manner. For a task this behaviour would usually consist of performing some kind of calculation based on the input, be it using machine or human computation, and then sending the result as output to some other task. For an operator, the behaviour would usually consist of a simple redistribution of the input to other operators, without actual computation done in the data. This highlights the role of operators as structural influence, rather than computational influence in the process. To simplify: *Tasks compute data, operators distribute data.*

Using this model we initially described components that are connected to each other in a sequential way. This seemed only natural, as a process is usually conceived as a "series" of actions. So we started building sequential webs of components, where each individual task would know its predecessor(s) and its successor(s).

The model was developed to a working prototype and implementation of some algorithms started. However, some problems were soon discovered with the approach.

3.6 Issues with the Sequential Approach

3.6.1 Non-local Dependencies and Information Propagation

The first issue with the above approach is the lack of global knowledge. What do we mean by that? Well, each actor contains certain information, e.g. what type of

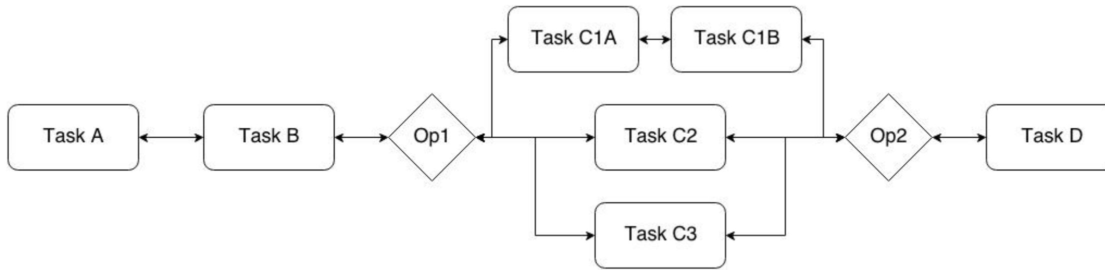

Figure 3.1: An Example of a Purely Sequential Process Model

component it is, what state it currently is in, what data it received, what results it computed using this data and whether it is still functional. On a local level, this data can be used to decide the correct course of action for every individual component and its predecessors and successors. On a non-local level however, this data is not available to every component in the process, which can lead to problems. Lets for example consider the case of *Op2* in Figure 3.1. It is waiting for the branch of Task C1A and Task C1B to finish before it can forward the sequence flow to Task D. Let us now assume that Task C1A fails during execution. In this case, Op2 will wait forever because it does not know about the failure. Task C1A knows it has failed, and through its connections Op1 and Task C1B also know about the failure. But Op2 has no direct connection to Task C1B, yet it is still dependent on its proper execution. This problem could potentially be solved using timeouts, although that is often not the best idea in the presence of human computation which often takes hours to complete. Alternatively, the problem requires some form of progagation of knowledge along the network, but we have to be careful not to broadcast irrelevant information to the entire network. In the example, neither Task A nor Task B have any interest in knowing that Task C1A has crashed. What we have to develop is a model that incorporates dependencies between tasks in order to send relevant information to the components that need it.

3.6.2 Dead-Ends

An different issue that comes up in the above example is the question of dead ends. Let's assume that in the figure the programmers accidentally forgot to connect Task C2 to Op2. In this case we have a dead end of computation. Task C2 will calculate a result to its subproblem, but no component will receive it. Somehow, we have to make sure that each solution for a subproblem is collected once again to solve the original problem statement. With a purely sequential approach, we cannot guarantee that each path of execution that has been opened at some point also gets closed again.

As a consequence of the above problems we decided that a purely sequential approach is not sufficient. Luckily, the Akka framework already gave us an idea that seemed more promising: a hierarchical structure of actors.

3.7 Hierarchical Approach

In Akka, each actor is part of a hierarchy. What is beneficial about this approach is, that each actor has some form of supervision. In case a child actor fails, its parent has the capability to detect this fault and restart the child actor or create a new one. It also has information about the progress its children are making and can forward this information to its own parent if so desired. This could potentially solve some of the shortcomings of the purely sequential approach, which is why we decided to pursue this concept for a while.

The most important question then becomes, whether it is possible to convert the sequential model described by Minder et. al into a hierarchical one. The following section will try to show that it is possible and how we accomplished that.

3.8 Mapping Sequential Patterns into Hierarchical Patterns

In this section we will try to show that it is possible to map sequential process models as described in the CrowdLang papers into hierarchical models. The very basic concept behind this transformation is to merge an opening operator like *Divide & Conquer* with its counterpart closing operator - in this case *Aggregate* - into a single operator that works as parent for the tasks that were between the two original operators. Let's consider the illustration in Figure 3.2.

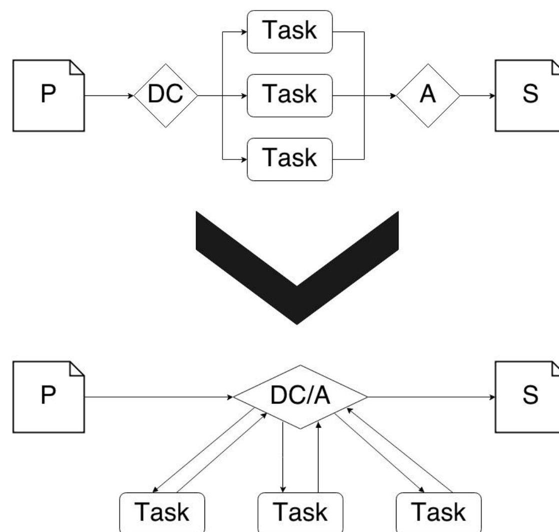

Figure 3.2: Conversion from a Sequential to Hierarchical Structure

As you can see, we merged the operators and attached the tasks as children just as mentioned above. This way we transformed the original sequential structure into a hierarchical one, with the added bonus of having a dedicated supervisory actor for the tasks. While this transformation is rather obvious for the above simple example, we want

to illustrate that it is also possible for more complex structures. Therefore we will later look at a more complicated example taken from [Minder and Bernstein, 2012a]. But first we need some names for the newly merged operators. We decided on the following:

Multiply and Merge/AND The definition of this pattern has slightly varied between the different papers about CrowdLang. The basic concept is that a problem statement is copied and distributed into multiple tasks and the solutions of said tasks are gathered into a set. In [Minder and Bernstein, 2012a] Minder et. al called this gathering *Merge* - to be distinguished from the subsequent *Reduce* operator, which selects the "best" solution from this set. In other papers, the *Merge* operator is replaced by an *AND* operator. This combination of operators has also been referred to as *Classical Collection* in some of the CrowdLang papers. To transform this pair of sequential operators into a single hierarchical operator we introduced our own ***Multiply*** operator.

Divide & Conquer and Aggregate This pattern is a fairly general one. It states that the original problem solution is divided into subproblems, however it does not specify how this division takes place. The most common use case is that the original problem is divided into multiple subproblems of the same kind. For this scenario we implemented the ***ForEach*** operator, which has two types of children. The first one is a single component that divides the problem into similar subproblems, whereas the second type is called for each of the subproblems generated. Eventually the ForEach operator collects all the solutions generated for the subproblems. The merging of the subproblems is left for a subsequent task.

XOR The XOR operator has the function to conditionally route the sequence flow in the sequential structure. We transformed this operator into the ***If*** operator. Additionally we provided an operator called ***Iterate*** which simplifies one of most common usages of the XOR operator, namely the iterative feedback loop where the solution of a task is fed back into the task until a certain condition is met.

This covers all the operators contained in the initial CrowdLang specification. However, since we abandoned the sequential approach, a new operator is necessary to indicate the sequence and data flow between two components. We called this operator ***Then***. A Then has two child components, where the first one accepts the input data and computes a result on it, which is then sent to the second child. This pattern introduces an extra layer of communication. Where in a sequential approach the first child would have simply sent its result directly to the second component, in the hierarchical approach this communication new runs through the parent Then operator.

Now that we have covered all operators of our language it is time to look at the more complex example of a human computation pattern as promised. We are therefore referring to the "Parallelized Interdependent Subproblem Solving" as described in [Minder and Bernstein, 2012a].

Figure 3.3 shows the pattern as described by Minder et. al.

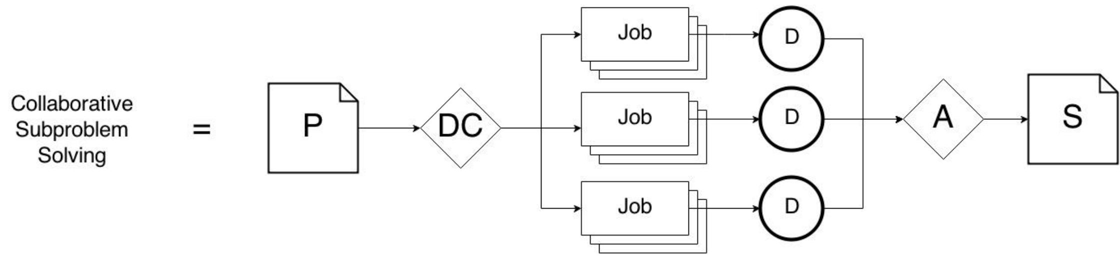

Figure 3.3: Collaborative Subproblem Solving by Minder et. al

The pattern on the first look seems similar to what we have shown above in our simple example, but it has to be noted that here both the *Job* as well as the *Decision* are patterns themselves, rather than atomic task. It is this composition that makes this pattern more complicated. In our hierarchical approach this pattern will have multiple levels in the hierarchy, so let us look at how we would construct it (Figure 3.4).

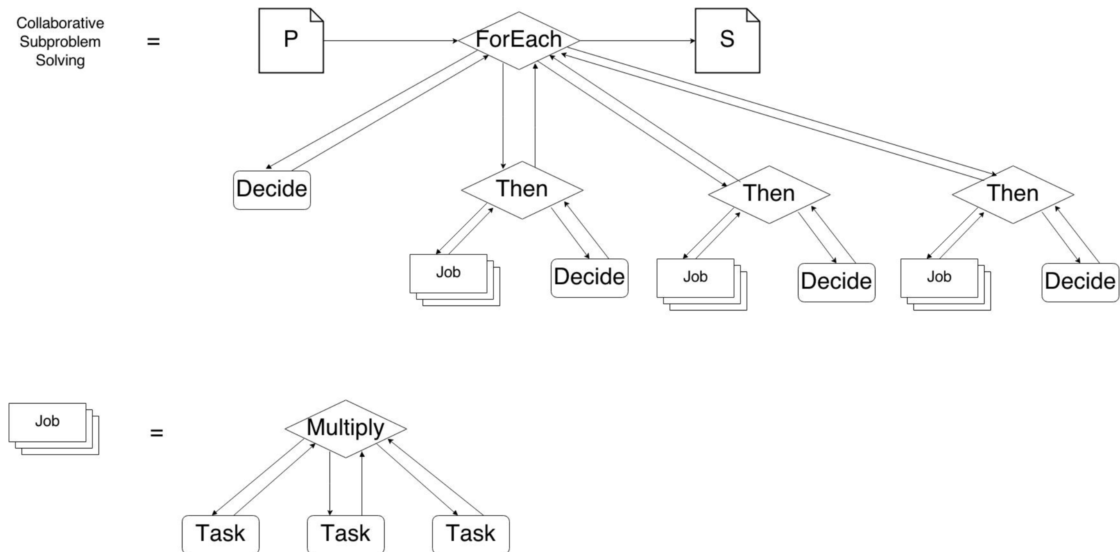

Figure 3.4: Collaborative Subproblem Solving Transformed

As you can see, the process is built in tree form. To indicate the sequential flow from the *Job* to the *Decision* components, we had to connect them with a Then operator. The root operator is a ForEach. As mentioned earlier, this operator has two types of children. On the left we have a *Splitter*. This can be any kind of Component that decomposes the initial problem into subproblems. These subproblems are then each sent to a copy of the other type of child components. This example shows that using our newly introduced operators, we can transform even complex interaction patterns from CrowdLang syntax into our own syntax. As we believe our hierarchical approach has substantial advantages

over a sequential approach, we will be following it in our implementation.

3.9 Summary

In this chapter we have shown that a purely sequential approach to modelling human computation processes has issues. We indicated that most of these issues can be solved when moving to a hierarchical approach instead. We then showed how sequential patterns can be transformed into hierarchical patterns using a simple example as well as a more complex pattern called "Parallelized Interdependent Subproblem Solving" that had previously been described by Minder et. al in [Minder and Bernstein, 2011]. Now that we have established our basic concept, we will go on to cleanly introduce and lay out our implementation of the hierarchical model. We will summarize the required components and show how they can be used and adapted with some code examples.

4

Implementation

In this chapter we are going to take a look at the implementation of our DSL. Our implementation is split into three parts. First of all, we will be discussing the core part which handles the basic components to be used in our language. The second part is concerned with the integration of crowd sourcing platforms and describes the API we provide as well as a sample implementation for the CrowdFlower website. The third part is concerned with triggering and handling events that allow future modules outside of the core structure (e.g. a GUI client) to gain information about the current status of the process network.

4.1 Akka in the Component Trait

This section discusses the core module of our DSL. We are going to start with a short description how Akka is used in our component trait and will then move on to an overview of the available operators and tasks including code snippets that demonstrate how they can be used. As mentioned before we used the basic actor model to develop our concept of a component trait, which summarizes the common requirements of tasks and operators.

The following snippet is part of the component class in our code.

```
trait Component extends Actor with ActorLogging {
  type Input

  /**
    * General behaviour that every component is able to perform
    * in every state
    */
  private def generalBevaviour : Receive = {
        ...
  }

  private val initial: Receive = {
    case Work(data: Input) =>
      log.info("Started work.")
      log.info(s"Data is: $data")
      work(data)
  }
  states.addState(initial, "initial")

  final override def receive = generalBevaviour orElse initial

 /**
   * Runs this component. Override to specify the behaviour
   * of your component when receiving Work.
   * @param data - the input data to work with
   */
  protected def work(data: Input)
  .
  .
  .
}
```

There are a few things to point out and explain in this code sample. Firstly, we want to take a look at the override of the receive method. As we have learned in previous sections, this method manages how an actor reacts to certain types of messages. It incorporates a partial function which is wrapped in a case class called "Receive". In this partial function, programmers can use pattern matching to filter messages received by another actor and decide on what to do with each type of message.

For our implementation, we want to make sure that each component is able to receive and respond to some control messages at any time, independent of the state it is in or the type of actor it is at runtime. Therefore we removed this source of customization by finalizing the receive method in our Component trait. We also used partial function chaining to make sure that the general behaviour is always part of an actor's receive method.

Another thing to point out is the *initial* behaviour. This is once again the same for each component. However, it is highly specifiable. The initial partial function reacts to any *Work* message containing data of the type *input*. This input type marks which kind of data an actor can actually handle and has to be specified by each concrete implementer of the Component trait. In response to the Work message, the component then calls the *work* method. What we achieve with this code is that we can execute some functions and methods any time a component receives a message, for example logging the message received, or throwing events for the event module. So, while this initial behaviour is the same for each component, the only thing it really constrains is the type of messages it can initially handle, namely messages of the *Work* type. How a component reacts to those messages is once again specifiable by each implementation of the *Component* trait via the *work* function.

We mentioned that one of the reasons to use the Akka framework is that its actors can encapsulate state information. At this point, said capability comes into play. After receiving and handling a Work message in the initial state as described in the previous paragraph, each component is free to change its behaviour and thereby the messages it accepts and handles. This incorporates a change of its internal state and it can be achieved via the state machine that has been defined within the component trait. The following code shows the method that can be used to manipulate said state machine.

```
/**
 * This function transitions the component to its next state.
 * Use it when all work is done for the current state.
 */
protected def nextState() = states.next()

/**
 * Signals that this component is done with work.
 * It enters the done state, where it can only process
 * messages defined in the general behaviour.
 */
protected def done() = states.done()

/**
 * Appends a new state to this components' state machine.
 * @param behaviour the behaviour that the actor should
 * exhibit in the new state
 * @param name a meaningful name to describe the state
 */
protected def addState(behaviour: Receive, name: String) =
        states.addState(behaviour, name)
```

The methods here are pretty straightforward. *nextState* transitions the component to its next state, while *addState* adds a new state to the end of the state machine. One thing to mention is the *done* state. In Akka, an actor that is done with its work would usually

get an order to shut down by its supervisor. Once it has shut down it will no longer be able to react to any messages. This behaviour is not desirable for our language, since we still want to be able to use the *general behaviour* mentioned above, even if the actor is done with its workload otherwise. This would for example allow the component to be reset to its original state, even after it had already done its work. Therefore we created the *done* state which is the final state of a component, where it waits until the process as a whole is complete and it finally receives a shut down order from its supervisor.

4.2 Operators

We will now take a look at the types of components we implemented. So let us quickly recapitulate from last chapter. Our language consists of *Tasks* and *Operators*. Tasks are computation units. They take an input and use it to calculate some sort of result. Operators provide structure to the data and sequence flow between tasks. Both of them extend the component trait. We have already implemented a series of operators that we used to replace the operators in CrowdLang. They are Then, Multiply, ForEach, If and Iterate. We will now look at how these operators can be used by programmers using simple code examples.

Then Then is the operator that indicates a sequence in the process.

```
val a = Props[A]
val b = Props[B]
val aThenB =  a.then(b)
aThenB ! Work(data)
```

The above code indicates that whenever a work request is sent to aThenB, it is first passed to a, whose result is then forwarded to b via the aThenB parent node. Once b is finished, its data is forwarded from aThenB to whoever sent the work request.

Multiply Multiply is the operator that indicates that each work request is copied n times and each copy is sent to a different child component.

```
val a = Props[A]
val multiply = a.multiply(10)
multiply ! Work(data)
```

In the code, when the multiply component receives work, it will spawn 10 child components of the type A. The data is then sent to each of these components and once they are finished processing, their results are gathered by the multiply component and forwarded to the original requester.

ForEach The *ForEach* operator has two child types. The first child splits the problem statement into subproblems. Therefore its result has to be of type *List*. Then, for each element in the list, a child of the second type is generated and processes the data.

Eventually, the results are gathered and passed to the original requester. The usage of this operator can be seen in the following code.

```
val splitter = Props[Splitter]
val b = Props[B]
val forEach = splitter.forEach(b)
forEach ! Work(data)
```

If The *If* operator functions as a conditional router in the process.

```
val ifActor = Props(new StringActor("If"))
val elseActor = Props(new StringActor("Else"))
val condition = (data : Any) =>
        data.asInstanceOf[String].equalsIgnoreCase("A")
val ifElseActor = ifActor.orElse(elseActor, condition)
ifElseActor ! Work("B")
```

The code snippet above shows how the *If* operator can be used. We define two tasks - in this case StringActors, who simply return the text provided to them on creation as result - and a condition for when to route to which task. In the provided example the ifElseActor would forward the traffic to the elseActor because the sent data "B" does not equal "A" , so we would get "Else" as result. Note that the function to use the If operator is called *orElse* since *if* is a reserved keyword in Scala.

Iterate The *Iterate* operator allows to repeatedly feed the result of a component back into a copy of itself. This allows for iterative improvement, which is a common pattern for crowd computing and also used in TurKit [Little et al., 2009]. The following snippet shows the usage of the *Iterate* operator.

```
val appendActor = Props(new AppendActor("B"))
val iterateActor = appendActor.iterate(4)
iterateActor ! Work("A")
```

The AppendActor is a simple actor that just appends the String provided at construction to the data it receives. This process is repeated 4 times, so the result of the above example will be "A B B B B".

All of the above operators have been implemented by creating a class that extends the *Operator* trait, which in turn extends *Component* trait that we mentioned earlier.

In order to use these operators in the form of functions, rather then having to create new objects every time we want to use them, we also introduced *TaskProps*. TaskProps is an implicit class that extends the normal functionality of the *Props* class that is required for actor creation in Akka with the above functions. So instead of writing code like this:

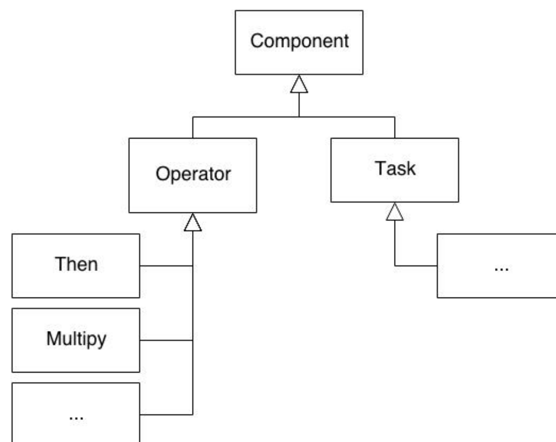

Figure 4.1: A Simplified Class Diagram of our Architecture

```
val a = Props[A]
val multiplyActor = new Multiply(childType = a,
    numberOfChildren = 10)
```

we can simply write code like this:

```
val a = Props[A]
val multiplyActor = a.multiply(10)
```

The TaskProps then handles the object creation.

Another thing to mention is that, while these operators should allow to implement most of the common and not so common interaction patterns in human computation processes, our code also allows the creation of additional, specialized operators. You can code your own operators by simply extending the *Operator* trait.

This concludes the section about operators. We have so far mostly talked about them, since they are the ones that provide structure in the web of human and machine computation tasks. But now it is time to also take a look at how tasks can be implemented using our language to perform some actual computation.

4.3 Tasks

While we can find a very small group of operators that will support most of the coordination patterns required for human computation processes, the variety in tasks descriptions is sheer endless. Of course there are well known default tasks like multiple choice questions or free text questions. But there are countless others that are very specific to the context of their application. Therefore the following section does not provide a list of the

tasks we programmed, but rather uses two examples to illustrate how one can implement their own versions.

We will start with a very simple example task, which is part of our *arithmetic* package. This package contains tasks that perform simple arithmetic operations on the data they are provided with and are mainly there to tie together more complex processes without having to break out of our language and doing the arithmetic operation manually.

The following code snippet shows the entire code of our *Addition* class.

```
class Addition extends Task {
  type Input = (Double, Double)

  /**
   * Runs this task. This message is called when a Task is
   * in its inital state and receive a Work message.
   * Adds the two numbers provided as input and
   * returns their sum.
   * @param data - the input data to work with
   */
  override protected def work(data: Input): Unit = {
    context.parent ! data._1 + data._2
    done()
  }

}
```

As you can see the Addition task has a very simple functionality. It expects input tuples of two Double values and returns their sum to the parent component. Then it goes into the *done* state. This task therefore only has two states, the initial state and the done state. It is an example for a machine computation task. One of the most important goals of our language is the seamless integration of machine and human computing. So let us next look at an example of a human computation task.

The *MultipleChoiceJob* task is part of our *hit* (Human Intelligence Task) package. This package contains default implementations for standard questions that almost every crowd computation platform supports. Currently it contains implementation for free text questions and multiple choice questions (which also supports single choice questions).

The following snippet contains the entire code for our *MultipleChoiceJob* class.

```
class MultipleChoiceJob(
        apicontroller: ActorRef,
        instructions: String,
        maxResults: Int = Integer.MAX_VALUE,
        options: Map[String, Any] = Map.empty[String, Any])
                        extends Task {
  type Input = List[String]
  /**
   * State 0 - send work to correct HIT platform
   * @param data- optional data to process
   */
  override def work(data: List[String]) = {
    val max = List(maxResults, data.size).min
    apicontroller ! MultipleChoiceString(instructions,
                            options = data,
                            maxNumberOfResults = max,
                            params = options)
    nextState()
  }

  /**
   * State 1 - wait for HIT platform to send results
   * then forward it to parent
   */
  private val waitForResult : Receive = {
    case MultipleChoiceStringResult(
                        choices : Map[String, Boolean],
                        work) =>
      context.parent ! choices
      done()
  }
  addState(waitForResult, "waitForResult")
}
```

The code is once again pretty simple. In the initial state, this task expects a list of String data, which are the options that should be provided for the multiple choice question. The task then simply wraps this data into a case class containing all necessary information for a crowd computing platform and sends it to the apicontroller and then goes to the next state. The *apicontroller* is a special actor within our network that accepts human computation task jobs and distributes them to the platforms. We will look at this in more detail in section 4.4. The *MultipleChoiceJob* task is now in the second state called *waitForResult*. It will stay alive and wait for a response from the *apicontroller*. It is worth mentioning that the task requires very little resources while waiting thanks to the Akka framework. Once the result from the crowd platform is

replied via the *apicontroller* it is simply forwarded to the parent component and the *MultipleChoiceJob* goes into its *done* state. As you can see this task has three states, but more states can easily be added by using *addState* method as mentioned earlier.

Our goal for all tasks was to keep the interface as flexible as possible to allow for any kind of task programmers might require. There are some default implementations available for a series of applications, like the arithmetic package or a package with tasks relevant to statistics. We hope that in the future this library of preprogrammed and reusable tasks can grow. Each algorithm implemented in our language will have potentially new sources of task configurations.

In the previous sections we have discussed the core building blocks of our language in operators and tasks. We will now go on to the second module of our implementation which is concerned with the integration of crowd computing platforms.

4.4 Crowd Computing Platform Integration

The crowd computing platform integration consists of two main parts: the *HITController* and the *HITWorkers*. The HITController is a dedicated actor that is known to all tasks that want to schedule jobs on a crowd computing site. We have already encountered this actor in the last section where we referred to it as *apicontroller*. Usually our network will have exactly one HITController for each website we want to interact with. If this ever becomes a bottleneck, it is easily possible to have multiple controllers per site though.

HITController The HITController expects any type of work that extends its *HIT-Work* trait and will create a new *HITWorker* for every job it receives. It will then wait for the HITWorker to return a result in the form of a *HITResult*. It will forward this result to the original requester. This class can usually be used unchanged for any integration of a new platform, since it is basically just acting as forwarding mailbox for a HITWorker.

HITWorker The HITWorker trait on the other hand is the main source of customization for any API integration. It uses partial function chaining in combination with a stackable trait pattern to allow any implementing platform to flexibly specify which types of jobs it supports from the existing library and also allows for additional custom jobs. We can use the *FreeTextHandler* as an example to show how this works.

```
trait FreeTextHandler extends HITWorker {
  receiver {
    case work: FreeText =>
      val future: Future[FreeTextResult] = writeText(work)
      future onComplete {
        case Success(r) => context.parent ! r
        case Failure(f) => context.parent ! Status.Failure(f)
      }
  }
  def writeText(work: FreeText): Future[FreeTextResult]
}

object FreeTextHandler {
  case class FreeText(
        instructions: String,
        params: Map[String, Any] = Map.empty[String, Any])
        extends HITWork

  case class FreeTextResult(text: String, work: FreeText)
        extends HITResult[FreeText]
}
```

The FreeTextHandler itself has to be a trait and extend the HITWorker trait to enable the stackable trait pattern [Odersky et al., 2008]. It declares in its *receiver* that it handles a new type of message called *FreeText*, which is defined as a case class extending the previously mentioned HITWork in its companion object. The partial function then calls the *writeText* method, which has to be overridden by any implementing worker. For example the *CrowdFlowerWorker* looks something like this:

```
class CrowdFlowerWorker extends HITWorker
with ActorLogging with MultipleChoiceStringHandler
with FreeTextHandler with MultipleChoiceImageHandler {
.
.
.
override def writeText(work: FreeText): Future[FreeTextResult]
        = scala.concurrent.Future {...}
.
.
.
}
```

As you can see, the programmer of the CrowdFlowerWorker has chosen to not only support free text jobs but also two kinds of multiple choice questions, one with text answer options and one with image answer options. If the CrowdFlower platform were

to introduce a new kind of job, the integration to our DSL could easily be adapted by creating a new handler like the FreeTextHandler and adding it to the traits implemented by CrowdFlowerWorker. These would be the only changes required in our API. In the case of CrowdFlower, the *writeText* by the way uses a the sites' RESTful API by preparing JSON requests and regularly querying the API for finished jobs.

4.5 Event Queue

The third module of our code is concerned with events within the network of components. While the networks' communication is implemented in the core module, some of the things that are going on while a computation process is running are of interest to third parties. We are, for example, currently working on a graphical user interface to work with our language. To support applications like these, we need to have some capabilities to transport data from within the network to this third party component. We decided to use an event based approach here, since it very well fits into the event based mechanisms that Akka already uses and is also standard for almost all model-view-controller architectures (MVC) out there. Our *EventQueueActor* can be freely added to any actor system created with our language. Similar to the HITController from the previous section, this actor is known to all of our components and whenever a component performs an action that should trigger an event it will send that event to the EventQueueActor, which will in turn distribute it to any third party that is interested in these kinds of events by using a publish-subscribe pattern.

This module is only in its infancy by now. As we mentioned we are working on a graphical user interface, but we are currently early on in development. Therefore this module is still subject to change and will probably evolve within the next months of work.

This concludes our implementation chapter. We have shown how we implemented the design we developed in earlier chapters and we have presented some examples on how programmers can use our language. We will now move to the evaluation part of this thesis were we implement two algorithms as proof of concept.

5

Evaluation

5.1 Implemented Algorithms

To show the capabilities of our programming language we implemented a series of examples, ranging from very simple examples only using computer power to complex algorithms that include human intelligence tasks deployed on CrowdFlower. Some of the basic examples have been used in the previous chapter for illustration purposes, the rest of them can be found in our codebase. In the following section we are going to show two examples that are more complicated and include human intelligence tasks deployed on CrowdFlower: text correction with Find Fix Verify and image categorization with confidence levels.

5.2 CrowdFlower Integration

For the deployment of our sample algorithms we chose the CrowdFlower platform. It provides a RESTful API to developers, who want to automatically schedule now jobs. In our experience, the API is powerful and well documented, but it is also very unreliable. Sending equivalent requests may often cause differing responses. Requests would sometimes time out, and other times the page would try to forward to another page (HTTP Code 302). Additional issues were caused by the lengthy process required to set up a job. The normal process for interacting with the API involved four steps. In the first step, a new job can be created using a POST request that includes URL based parameters. In the second step, data has to be provided for the job. As an example: to create a multiple choice question, one would first have to create a new multiple choice type job, and then send the answer options in the second step as POST request. This data has to be formatted in JSON. In the third step CrowdFlower expects a launch command provided via HTTP POST, but this time in URL encoding again. After that, the API can continuously be queried with GET requests, to find out whether a job has been completed. This complex process, in combination with the unreliable responses, made it quite hard to interact with the API and we hope that future versions can improve.

After a large amount of debugging and trial and error, we were able to address the above issues in our code. We are for example resending requests that were not handled

correctly and we are limiting the amount of requests that can be sent at once, so that one does not overload the CrowdFlower API.

After having discussed the struggles we encountered during implementation, we will now come to the description of the algorithms we ran on CrowdFlower, starting with Find Fix Verify.

5.3 Find Fix Verify

One of the sample applications developed in our language is the Find-Fix-Verify pattern as it was described in the Soylent paper [Bernstein et al., 2010]. It conceptualizes an algorithm to be used for any kind of quality improvement process. Such processes have been addressed in other crowd computing algorithms as well, including ones where multiple improvement steps have been chained together to iteratively improve the solution for a given problem like TurKit [Little et al., 2009]. The contribution of the Find Fix Verify algorithm in this context is the decomposition of the task into smaller subtasks. The motivation for this deconstruction is the authors' observation that crowd work comes in a wide variety of qualities. "We are primarily concerned with tasks where workers directly edit a user's data in an open-ended manner. [...] In our experiments, it is evident that many of the raw results that Turkers produce on such tasks are unsatisfactory. As a rule-of-thumb, roughly 30% of the results from open-ended tasks are poor." [Bernstein et al., 2010] The authors describe their algorithm on the application of shortening a written paragraph. Workers are provided with a long section written by a user of the Soylent Microsoft Word plug-in and are expected to shorten the section. In order to combat the cited problems, they split the task into three stages called *Find*, *Fix* and *Verify*. We will now take a look at how the three steps work exactly.

5.3.1 Concept

Find

The first step in the process is the *Find* step. At the beginning, the long section is split into paragraphs. Workers in this step are then asked to identify only those paragraphs that provide room for shortening the section as a whole. This step can be modelled as a multiple-choice task, where each part (paragraph) of the solution is one of the choices. Just like subsequent steps, this first step is performed by multiple workers in parallel. Among all the paragraphs marked to be too long - we call them "patches" - only those that have been mentioned by at least 20% of all workers are sent to the next step.

Fix

In the Fix step, the workers are asked to actually improve on the patches selected in the Find step. They see the original paragraph and their job is to shorten it without losing information in the process. Once again, this task is performed by multiple workers in parallel to get a decent solution space.

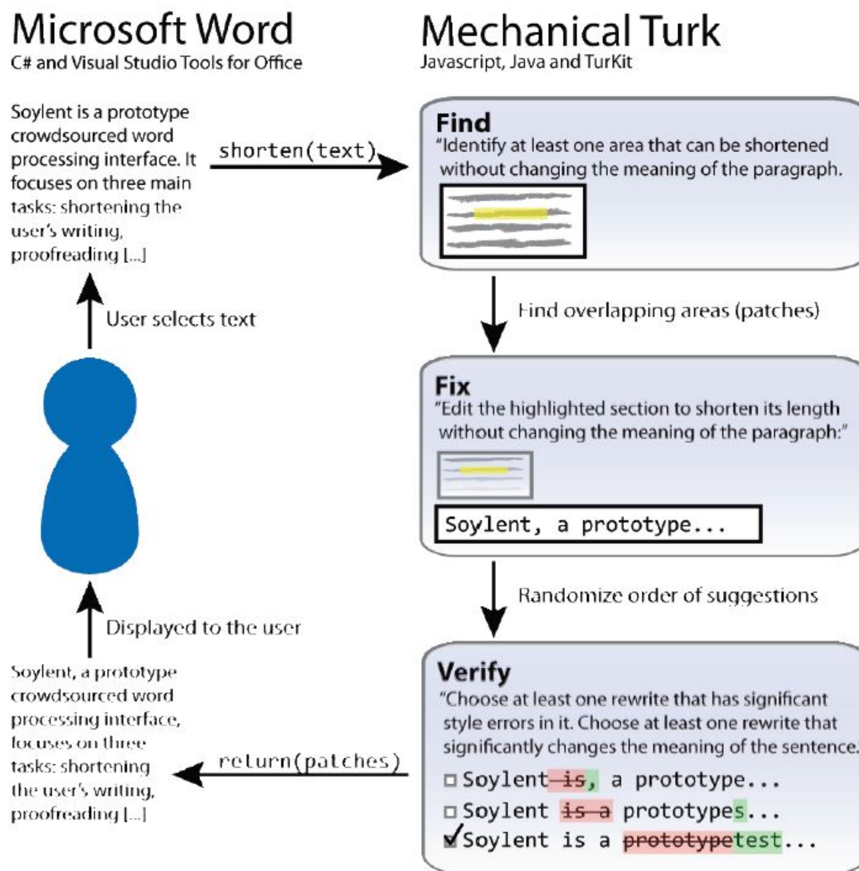

Figure 5.1: The Find Fix Verify Algorithm in Shortn

As mentioned, by separating the Find and Fix stages the authors of the Soylent paper try to combat the lazy workers' tendency to only focus on the simplest paragraph to shorten. When the workers who identify the paragraphs with potential are separated from those who have to shorten them, lazy workers are forced to take care of more difficult tasks as well.

Verify

In the Verify step, the previously gathered solutions are put to a contest for quality control. Workers in the Verify step are presented a single-choice question with all the proposed improvements as well as the original passage for reference. After all workers have voted on what they think is the best improvement, the proposal with the most votes is selected and integrated into the original text. According to the authors, the Verify stage "[...] reduces noise in the returned result."[Bernstein et al., 2010]

5.3.2 Experimental Setup

While the authors of the Soylent paper chose paragraph shortening as an illustrative example, we decided to go with the other algorithm mentioned in their framework for our experimental application, namely proofreading. This section will show the exact workings of our sample application, including screenshots from the CrowdFlower jobs presented to workers.

Find

In the Fix step of our process, workers are presented with a multiple choice task. They are given an entire paragraph containing grammatical errors and are asked to select those sentences, which they think will need corrections. Note here, that users can and are encouraged to select multiple options. Figure 5.2 shows a sample task for a worker in the Find stage.

Find grammatical errors in a text passage.

Instructions ▴

The following paragraph contains grammatical errors. Please identify and select all sentences that need corrections. NOTE: Please note that this task is part of a one time expirement and you are not required to fill out the feedback form for the task.

Select ALL sentences that require work

- [ ] Jaime been applying for full-time jobs for several months.
- [ ] The last week he received a call from the Human Resources director at a computer software company the HR director asked Jaime could he fly to Chicago for a job interview.
- [ ] The company offered to pay for Jaime's plane ticket to Chicago, so that he will not have to pay for it himself.
- [ ] Jamie agreed to come for the interview.
- [ ] Since then, Jaime has been busy collecting informaton about the company.
- [ ] He also went shopping for a new suite.
- [ ] Jaime and his wife have been rehearsing answers to possible interview questions, so that Jaime will be good and ready his best.
- [ ] Jaime is nervous about the interview, but his looking forward to working at a new place.

The following paragraph contains grammatical errors. Please identify and select all sentences that need corrections. NOTE: Please note that this task is part of a one time expirement and you are not required to fill out the feedback form for the task.

Figure 5.2: Sample Task of the ”Find” Stage in CrowdFlower

All stages of the FFV algorithm should be completed multiple times and in parallel by different workers. From their suggestions, we then automatically selected those answers, that had been marked by at least 20% of the them. For each of the selected sentences, we queued up a number of Fix jobs on CrowdFlower.

Fix

In the Fix stage, the workers are presented with one of the previously identified sentences and are asked to fix all grammatical errors they can find. In order to do so, they have a text field that allows for arbitrary input. Figure 5.3 shows a sample task for a worker in the Fix stage.

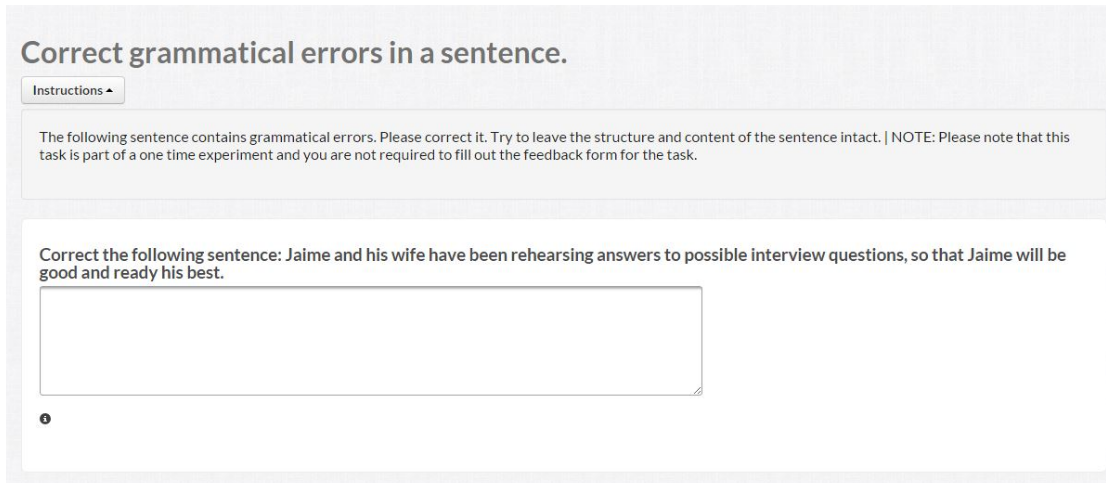

Figure 5.3: Sample Task of the "Fix" Stage in CrowdFlower

The results that are acquired here are expected to be of very variable quality as has been mentioned before. Therefore, all solutions for a specific sentence are gathered and presented in the Verify step, where another group of workers can then vote on which solution they deem to be the best improvement.

Verify

Figure 5.4 shows a sample task for a worker in the Verify stage, where they are asked to select the best solution among those gathered in the Fix step.

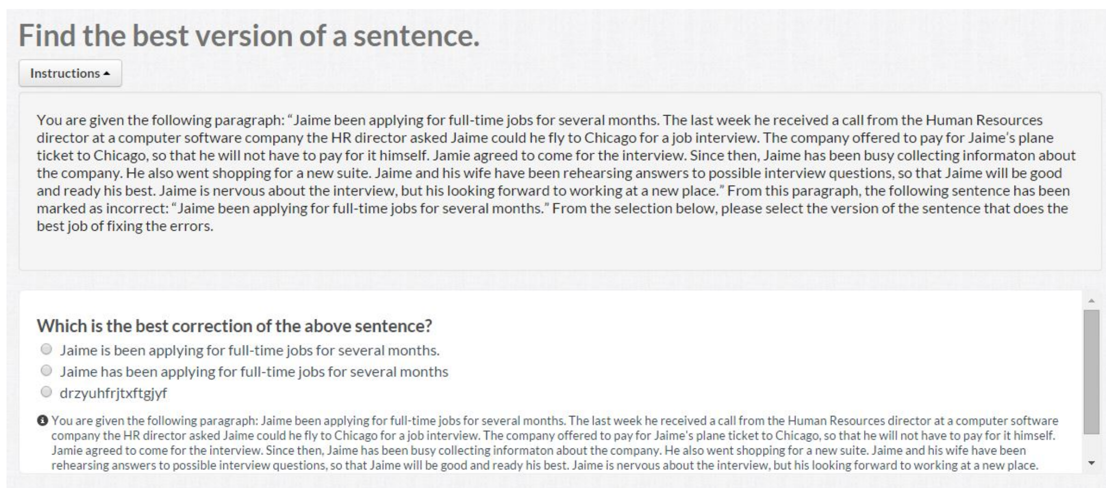

Figure 5.4: Sample Task of the "Verify" Stage in CrowdFlower

Once all the tasks in the Fix and Verify stages have been completed, the best solutions

are then gathered and merged together with the original, now improved paragraph.

5.3.3 Results

For the evaluation we ran the algorithm multiple times with different specifications on the amount of workers involved. Bernstein et al. suggested 3-5 workers for each the Fix and Verify stages of the algorithm. For the Find stage they made no clear suggestion. We therefore executed the algorithm in three configurations:

- 2 workers in the Find stage, 3 per Fix, 3 per Verify
- 5 workers in the Find stage, 3 per Fix, 3 per Verify
- 10 workers in the Find stage, 5 per Fix, 5 per Verify

All workers received a compensation of $0.10, no matter in which stage or which run they participated in.

Let us look now at the following paragraph that has been entered into the algorithm during our evaluation phase. It is a sentence from a website [English for Everyone, 2014], which lets users work on their grammar skill. It does therefore contain several grammatical errors on purpose, which should be corrected in our process.

Jaime been applying for full-time jobs for several months. The last week he received a call from the Human Resources director at a computer software company the HR director asked Jaime could he fly to Chicago for a job interview. The company offered to pay for Jaime's plane ticket to Chicago, so that he will not have to pay for it himself. Jamie agreed to come for the interview. Since then, Jaime has been busy collecting informaton about the company. He also went shopping for a new suite. Jaime and his wife have been rehearsing answers to possible interview questions, so that Jaime will be good and ready his best. Jaime is nervous about the interview, but his looking forward to working at a new place.

What we found in our experiments was, that the results of our algorithm heavily depended on the amount of workers involved in the Find stage. Bernstein et. al had the goal to reduce the amount of bad results that make it to the end of the process and hoped to achieve it by including the Verify stage and having multiple people perform the Fix stage at once as quality control mechanisms. This goal was reached - at least in our experiments. But the authors of Soylent also hoped to reduce the influence of lazy workers in the algorithm by splitting the Find and Fix stages. In their words: "Lazy Turkers will always choose the easiest error to fix, so combining Find and Fix will result in poor coverage. By splitting Find from Fix, we can direct Lazy Turkers to propose a fix to patches that they might otherwise ignore." [Bernstein et al., 2010]. We believe this goal has only partially been reached. While it is true, that lazy workers are forced to also correct harder mistakes in the Fix stage due to the setup, their influence in the Find stage on the other hand is very strong. In the environment of a combined Find

and Fix stage, lazy workers will only *fix* the easiest mistake. In the environment of split Find and Fix stages, they will only *find* the easiest mistake. As mentioned, that made the algorithm highly dependent on the amount of workers in the Find stage. The more workers in the find stage, the better the chance of having multiple lazy workers "accidentally" find different problems (i.e. not every workers finds the same easy mistake, but rather another easy mistake instead).

The following paragraph shows the results we gathered in our first configuration with only two workers in the Find stage.

Jaime has been applying for full-time jobs for several months. The last week he received a call from the Human Resources director at a computer software company. The HR director asked Jaime could he fly to Chicago for a job interview. The company offered to pay for Jaime's plane ticket to Chicago, so that he will not have to pay for it himself. Jamie agreed to come for the interview. Since then, Jaime has been busy collecting informaton about the company. He also went shopping for a new suite. Jaime and his wife have been rehearsing answers to possible interview questions, so that Jaime will be good and ready his best. Jaime is nervous about the interview, but his looking forward to working at a new place.

Comparing this to the sample solution is rather disappointing:

Jaime has been applying for full-time jobs for several months. Last week he received a call from the Human Resources director at a computer software company. The HR director asked Jaime if he could fly to Chicago for a job interview. The company offered to pay for Jaime's plane ticket to Chicago, so that he will not have to pay for it himself. Jamie agreed to come for the interview. Since then, Jaime has been busy collecting information about the company. He also went shopping for a new suit. Jaime and his wife have been rehearsing answers to possible interview questions, so that Jaime will be prepared. Jaime is nervous about the interview, but he is looking forward to working at a new place.

As you can see, only one mistake was actually fixed. That is the case, because it was the only sentence marked for correction in the Find stage. In this context, it is also not surprising that it was the first sentence in the paragraph that was up for selection in the Find task. The workers involved probably read the task description and then immediately found the (relatively) easy mistake in the first paragraph, which prompted them to select it and commit their result.

For comparison, the following paragraph is the solution of the run-through that included 10 workers in the Find stage. As you can see, the results are much better here, although the text is still far from perfect and some of the corrections are not really of high quality. A total of six sentences were selected in the Find stage compared to the one sentence in the run above.

Jaime has been applying for full-time jobs for several months. Last week he received a call from the Human Resources director at a computer software company the director asked Jaime could he fly to Chicago for a job interview. The company offered to pay for Jaime's plane ticket to Chicago, so that he will not have to pay for it himself. Jamie agreed to come for the interview. Since then, Jaime has been busy collecting information about the company. He also went shopping for a new suit. Jaime and his wife have been rehearsing answers to possible interview questions, so that Jaime will be prepared. Jaime is nervous about the interview, but he is looking forward to working at a new place.

The apparent dependency on the amount of workers in the Find stage is an interesting observation that could be investigated in more detail in the future. Especially when considering to use the FFV algorithm in a productive environment where compensation is a cost factor, finding the optimal amount of workers required to get decent results for a given task is an important question. For our own results, we had no problem increasing the amount of workers involved, as we only ran the algorithm a few times. The only trade-off we had to make was an increased time commitment to complete the algorithm. But both, the time commitment as well as the compensation can become important factors when massive amounts of tasks have to be computed for a company.

5.4 Image Labelling

5.4.1 Concept

For our second use case we decided to go with an image categorization task. The exact idea stems from the Automan [Barowy et al., 2012] paper and is there referred to as "Which one does not belong?". Users are confronted with a selection of images among which they have to find the one that does not belong with the others. This kind of task is often part of building ontologies and clusters of images. What made this use case interesting for us specifically is, that on top of the relatively simple single choice question, the authors of Automan extended the process to calculate the statistical measure of confidence levels after each set of jobs for a task to conditionally terminate the algorithm or queue new jobs. This element of conditional task deployment is relatively rare and not used or mentioned in the other papers we studied. It is therefore an interesting challenge for our language.

5.4.2 Experimental Setup

We chose to go with cars as the subject of our categorization experiment. We provided each user with five images of cars, where in each case four of them were of the same manufacturer whilst the other was of a different manufacturer. Users were then asked to find the one that does not belong to the others. So the question is a single choice question. Each worker was paid $0.10 to complete this task.

Figure 5.5: Example Selection of Cars

On the first run three jobs were scheduled at once, since this is the minimum number of people that have to agree for the required 90% confidence. Once the first set of workers finished the task, the results were compared and confidence levels were calculated. If all three workers had agreed, the confidence level was obviously reached and the result was returned. If there was disagreement, we found the maximum amount of agreeing workers and used this number to determine how many new jobs had to be at least scheduled to reach the required confidence level in case the new workers all agreed with the previous majority. This is what the authors of Automan called the optimistic approach. The confidence level calculations were implemented using MonteCarlo simulation, just like the authors of Automan did in their public github repository [Barowy, 2014].

5.4.3 Results

As we implemented this algorithm mainly as proof of concept, the number of runs is once again limited. We ran it a total of five times, with a different selection or ordering of cars each time. In all our executions, the results gathered showed that workers were able to correctly identify the car that did not fit with the others. For example in 5.5 the workers decided on the first car from the left as the outsider, which is the correct choice. This car is an Audi, whilst the other cars are all BMWs. Interestingly, this exact run had a very special outcome. Many Workers named the second car from the left instead of the Audi. This was probably the case because the image shows an older model than the other cars in the selection and therefore looks slightly different. Unsurprisingly, runs of the algorithm including this image required the most workers to reach the confidence level of 90% that we aimed for. They required 11 , 9 and 8 workers in total, whereas the other runs required 5 and 6 workers respectively.

5.5 Discussion

In the previous sections we have implemented two algorithms using our domain specific language and CrowdFlower as a source for human computation. We have discussed the problems of the CrowdFlower API and how we addressed them. We then saw, how the Find Fix Verify algorithm is very susceptible to lazy or malicious workers in the Find stage, which led to disappointing results when running the algorithm with only two workers in the Find stage. We pointed out that there might be potential for future research in this area. Meanwhile, our image categorization task ran without problems

and the workers were consistently capable of identifying the car which did not belong to the others.

In any case, the main goal of the evaluation section was not to measure the performance of the algorithms themselves, but rather to show that our language can be used to develop fully integrated computation algorithms. The evaluation part is therefore to be viewed mainly as a proof of concept. And after some initial struggles with the CrowdFlower API we reached our goal.

A limitation to the results presented is, that all experiments have been conducted on a single crowd computing platform. This leaves open the question whether the existing code base is sufficiently flexible as well as powerful for the seamless integration of different websites such as Amazon's Mechanical Turk.

It is also worth mentioning that most of the programming and debugging for the Find Fix Verify and image categorization algorithms has gone hand in hand with the further development of both the DSL, as well as the CrowdFlower API integration. It is therefore yet to be seen whether the DSL is intuitive and flexible enough to support the development of different algorithms by personnel outside the original developer team.

Furthermore, a side concern for our DSL was the eventual possibility of distributed computing. That side concern was part of the reason why we decided to use Akka as a foundation for our implementation, but although this foundation should be a reasonable starting point for distributed computing, it has yet to be shown whether we can handle massive amounts of components distributed over a cluster of processors. This concern will once again be taken up in our suggestions for future work.

6

Future Work

The goal of this thesis was the initial description and implementation of a human computation programming language based on the concepts of CrowdLang. That being the case, the possibilities for future work based on the our current results are manifold.

A first approach to extending the current environment lies in the implementation of further algorithms using the language. As proof of concept, we implemented two well-known algorithms in Find-Fix-Verify and image categorization with confidence levels. Additionally we included an implementation of an algorithm based on a dual pathway structure [Liem et al., 2011] as part of our development process, although we have not evaluated that algorithm yet. It is evident, that there exist a variety of other human computation algorithms which can be programmed in our language - after all, that was the main motivation for our project.

Another branch of future work should be the implementation of other operators and tasks. Our current code covers all the concepts described in the CrowdLang papers, and we are confident that with these operators alone a vast variety of human computation processes can be covered. However, it is to be expected that additional operators or specializations and adaptations of existing ones for different applications will be required. Of course this development goes hand in hand with the development of additional human computation processes in our language as described above. Whenever the existing set of operators are not enough to implement a certain process, we wish to extend the set of operators. The same goes for new, flexible implementations of tasks. Our goal is to eventually have an extensive library of reusable tasks and operators in our language.

A more technical area of future work is the integration of more crowd computing websites with our language. For our evaluation we implemented an integration with CrowdFlower, but at all times tried to keep the interface as flexible as possible to allow for future integration of additional websites and applications like Mechanical Turk or ClickWorker. We hope that with growing interest in our language, the number of available platform integrations will also increase.

Furthermore, our language has from early on been built with distributed computing in mind. While current algorithms for human computation do not explicitly require a large amount of computing power in terms of hardware, the better the integration of human work and computers becomes, the more we have to expect the need for distribution of computing power onto multiple cores. As our language is built on the Akka framework,

we would expect that from a conceptual perspective it is already very suitable for distributed computing. However, the proof of this hypothesis is one to be made. Additional research and improvements in this area are to be expected.

7

Conclusions

In this thesis we presented our new domain specific language based on the concepts of CrowdLang. We started by describing the challenges that current crowd work faces. While simple, atomic tasks can readily be deployed and solved using platforms like Amazon's Mechanical Turk, more complex problems that need coordination and data flow between the individual tasks are still hard to solve. We showed that the need for coordination has been addressed by a multitude of research groups and that some patterns as well as frameworks to implement them have been proposed. We discussed how CrowdLang offers a more open framework than other comparable papers who solely focus on one type of interaction, for example CrowdForge with the Map-Reduce paradigm.

Starting from CrowdLang we then initiated the development of our own model for a possible implementation of a domain specific language. We found that a sequential approach, while intuitively understandable and well suited to graphically represent a process, is not a preferable structure for the actual implementation of our language. When individual tasks are just sequentially connected to each other, it is hard to keep track of dead-end branches in the network and establishing global knowledge about the state of the process is difficult. We therefore worked towards a hierarchical model, where a splitting operator and a merging operator from the sequential model are merged into a single parent operator in the hierarchical model. All components in between the two sequential operators are then attached as child nodes to the hierarchical operator. We showed how one can transform coordination patterns in the sequential model into equivalent patterns in the hierarchical model.

Once we established our concept for the domain specific language, we showed how we implemented it using the Akka framework and its actor model. We discussed how a *Component* trait can be used to incorporate the similarities between operators and tasks. We pointed out details in our implementation and showed some examples on how to use the language. What makes our language so powerful is the enormous flexibility it offers to users. While we established a set of default operators in *Multiply*, *Then*, *ForEach*, *Iterate* and *If* that should be sufficient to implement most algorithms in the crowd sourcing environment, we also showed how one can program their own dedicated operators. Thereafter we explained how one can specify and use their own tasks. As tasks are very specific to their use case, we decided to leave their implementation open to developers.

In the remainder of the implementation chapter, the modules for event handling and crowd computing platform integration were introduced. It was shown how one can use the interface to accommodate for any crowd computing platform out there, although we have only provided a sample integration for CrowdFlower so far.

We used said CrowdFlower integration in our evaluation section to run two different algorithms as a proof of concept. The first algorithm we ran, was an adaptation of the *Find-Fix-Verify* algorithm introduced in Soylent [Bernstein et al., 2010]. We provided CrowdFlower workers with a text passage that contained grammatical errors and over the three stages of the algorithm and the contribution of more than twenty workers, we were able to fix at least some of the mistakes. But we also noticed that the lazy worker problem could not be solved completely with this algorithm, as the influence of lazy workers in the find stage negatively influenced the whole process. We found that when using only a small number of workers in the Find stage, the results acquired are drastically reduced in quality.

In our second algorithm for the evaluation part, we implemented a image categorization task, where users were asked which one of a set of cars does not belong to the others (the criterion for exclusion was brand). In this algorithm we continuously put additional jobs onto CrowdFlower, asking the same question, until a sufficient number of workers had agreed on one of the options for us to expect the correct result with a 90% confidence level. Our observations showed that workers here were able to identify the correct answer in all of our executions after 11 scheduled jobs at most.

Following our evaluation, we pointed out limitations to our results. The language has yet to be tested by developers outside of the original developer team. Also, we have so far only implemented an integration for one crowd platform - namely CrowdFlower. We pointed out these and other angles for further research and development in our future work section, which concluded this thesis.

References

[Ahmad et al., 2011] Ahmad, S., Battle, A., Malkani, Z., and Kamvar, S. (2011). The jabberwocky programming environment for structured social computing. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 53–64, New York, NY, USA. ACM.

[Akka, 2014] Akka (2014). Actor systems. *http://doc.akka.io/docs/akka/2.3.6/general/actor-systems.html*.

[Armstrong, 2007] Armstrong, J. (2007). A history of erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 6–1–6–26, New York, NY, USA. ACM.

[Barowy, 2014] Barowy, D. W. (2014). Automan github repository. *https://github.com/dbarowy/AutoMan*.

[Barowy et al., 2012] Barowy, D. W., Curtsinger, C., Berger, E. D., and McGregor, A. (2012). Automan: A platform for integrating human-based and digital computation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 639–654, New York, NY, USA. ACM.

[Bernstein et al., 2010] Bernstein, M. S., Little, G., Miller, R. C., Hartmann, B., Ackerman, M. S., Karger, D. R., Crowell, D., and Panovich, K. (2010). Soylent: a word processor with a crowd inside. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, pages 313–322. ACM.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

[English for Everyone, 2014] English for Everyone (2014). Paragraph correction. *http://www.englishforeveryone.org/Topics/Paragraph-Correction.htm*.

[Kittur et al., 2013] Kittur, A., Nickerson, J. V., Bernstein, M., Gerber, E., Shaw, A., Zimmerman, J., Lease, M., and Horton, J. (2013). The future of crowd work. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, CSCW '13, pages 1301–1318, New York, NY, USA. ACM.

[Kittur et al., 2011] Kittur, A., Smus, B., Khamkar, S., and Kraut, R. E. (2011). Crowdforge: Crowdsourcing complex work. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 43–52, New York, NY, USA. ACM.

[Liem et al., 2011] Liem, B., Zhang, H., and Chen, Y. (2011). An iterative dual pathway structure for speech-to-text transcription. In *Human Computation*.

[Little et al., 2009] Little, G., Chilton, L. B., Goldman, M., and Miller, R. C. (2009). Turkit: tools for iterative tasks on mechanical turk. In *Proceedings of the ACM SIGKDD workshop on human computation*, pages 29–30. ACM.

[Malone et al., 2010] Malone, T. W., Laubacher, R., and Dellarocas, C. (2010). The collective intelligence genome. *Sloan Management Review*, 51(3):21–31.

[Minder and Bernstein, 2011] Minder, P. and Bernstein, A. (2011). Crowdlang - first steps towards programmable human computers for general computation. In *Human Computation, Papers from the 2011 AAAI Workshop, San Francisco, California, USA, August 8, 2011*.

[Minder and Bernstein, 2012a] Minder, P. and Bernstein, A. (2012a). Crowdlang: A programming language for the systematic exploration of human computation systems. In *Proceedings of the 4th International Conference on Social Informatics*, SocInfo'12, pages 124–137, Berlin, Heidelberg. Springer-Verlag.

[Minder and Bernstein, 2012b] Minder, P. and Bernstein, A. (2012b). Crowdlang: programming human computation systems. Technical report, University of Zurich.

[Minder and Bernstein, 2012c] Minder, P. and Bernstein, A. (2012c). How to translate a book within an hour: Towards general purpose programmable human computers with crowdlang. In *Proceedings of the 4th Annual ACM Web Science Conference*, WebSci '12, pages 209–212, New York, NY, USA. ACM.

[Noronha et al., 2011] Noronha, J., Hysen, E., Zhang, H., and Gajos, K. Z. (2011). Platemate: crowdsourcing nutritional analysis from food photographs. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 1–12. ACM.

[Odersky et al., 2008] Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala*. Artima Inc.

[Quinn and Bederson, 2009] Quinn, A. J. and Bederson, B. B. (2009). A taxonomy of distributed human computation. *Human-Computer Interaction Lab Tech Report, University of Maryland*.

[Raychaudhuri, 2013] Raychaudhuri, N. (2013). *Scala in Action*. Manning Publications Co.

List of Figures

3.1 An Example of a Purely Sequential Process Model 13
3.2 Conversion from a Sequential to Hierarchical Structure 14
3.3 Collaborative Subproblem Solving by Minder et. al 16
3.4 Collaborative Subproblem Solving Transformed 16

4.1 A Simplified Class Diagram of our Architecture 24

5.1 The Find Fix Verify Algorithm in Shortn 33
5.2 Sample Task of the "Find" Stage in CrowdFlower 34
5.3 Sample Task of the "Fix" Stage in CrowdFlower 35
5.4 Sample Task of the "Verify" Stage in CrowdFlower 35
5.5 Example Selection of Cars 39