UNIVERSIDADE DO PORTO · FACULDADE DE ENGENHARIA

End-User Programming in Mobile Devices through Reusable Visual Components Composition

TIAGO MANUEL DA SILVA ALMEIDA

June 2012

Master in Informatics and Computing Engineering
Scientific Supervision by

Hugo Sereno Ferreira (PhD, Assistant Lecturer)
Department of Informatics Engineering

and Co-Supervisioned by
Tiago Boldt Sousa (MSc)

Contact Information:

Tiago Manuel da Silva Almeida
Faculdade de Engenharia da Universidade do Porto

Rua Dr. Roberto Frias, s/n
4200-465 Porto
Portugal

Email: tiago.silva.almeida@fe.up.pt

End-User Programming In Mobile Devices through Reusuable Visual Components Composition

Tiago Manuel da Silva Almeida

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Doctor João Carlos Pascoal de Faria
External Examiner: Doctor Angelo Manuel Rego Silva Martins
Supervisor: Doctor Hugo José Sereno Lopes Ferreira

July 17, 2012

…to Cris.

Abstract

An era where smartphones surpassed the PCs sales has begun [stPSi]. In this era, the end-users (the ones that ultimately use the smartphone) are demanding quantity and complexity from their applications and devices. This demand makes it impractical for a software developer to "foresee" every possible combination and explore every valid alternative.

A possible solution to make customizable applications is to empower end-users with end-user programming (EUP) tools. EUP is "the practice by which end users write computer programs to satisfy a specific need, but programming is not their primary job function" [LSBM09]. That EUP tool could be a collaborative framework, where novices and experts can co-exist and share their implementations, while end-users explore their requirements. With such scenario, expert programmers can create components that end-users can connect using a *visual programming language* (VPL). Such tool could not only reduce the number of "small", similar, specific-tailored applications, but also foster discovery and experimentation by end-users.

To construct this tool we start with an analysis into the problems we have to solve. Studies in the EUP gave us a start with six recurring problems end-users have [KMA04]. Next, we went further and explored another area: End-User Software Engineering (EUSE). EUSE tries to embed software engineering concepts into EUP solutions [KAB+11]. Additionally we analyzed VPLs since they can be a good solution for EUP [WNF06]. This analysis was followed by a summary of the VPLs properties, and some examples of VPLs usage.

With the gathered knowledge, we created a prototype for Android to study how end-users percept and accept a VPL in a smartphone. In this prototype end-users can connect blocks and create tasks. Those tasks can be shared and reused, and they depended on events generated by the smartphone. With this prototype we tried to provide answers to questions such as: *Can we implement a VPL in a small screen? What is the necessary level of abstraction so we don't confuse end-users and we don't limit expert programmers? How can we integrate task reusing with block reusing without confusing end-users?*

We made several changes in our prototype because of end-users' feedback. However, when some of the improvements weren't done because we felt limited by Java's language. To surpass this limitation we conducted some experiments where we tried to combine Java with Prolog, and Java with Scala. Those experiments allowed us to create a rewrite rule system to automatically group blocks according to a set of rules defined by expert programmers.

Resumo

Uma era em que os smartphones superaram a venda de PCs começou. Nesta era, os utilizadores finais (aqueles que utilizam, em ultima instância, o smartphone) estão a exigir quantidade e complexidade das suas aplicações e dispositivos. Esta exigência faz com que seja impraticável para um desenvolvedor de software, a previsão de todas as combinações possíveis e todas as alternativas válidas.

Uma possível solução para fazer aplicações costumizáveis passa por passar para as mãos dos utilizadores finais ferramentas que usam EUP (do inglês, "End-User Programming"). EUP "a escrita de programas de computador por parte de um utilizador final para satisfazer uma necessidade específica, mas a programação não é o trabalho principal deste utilizador final" [LSBM09]. Essa ferramenta que usa EUP poderia ser uma framework colaborativa, onde novatos e experientes poderiam partilhar as suas implementações e coexistir, enquanto os utilizadores finais exploram os seus requisitos. Com este cenário, utilizadores experientes podem criar componentes que podem ser ligadas por utilizadores finais com o uso de uma VPL (do inglês, "Visual Programming Language"). Esta ferramenta poderia reduzir o número de pequenas aplicações desenhadas para um problema muito específico, mas também poderiam impulsionar a descoberta e experiencias dos utilizadores finais.

Para fazer esta ferramenta começamos por analisar quais os problemas que tínhamos de resolver. Estudos no EUP deram-nos um começo com seis problemas recorrentes entre utilizadores finais [KMA04]. Depois, fomos mais além e exploramos outra área: EUSE (do inglês, "End-user Software Engineering"). EUSE tenta incluir conceitos de engenharia de software em soluções que usam EUP [KAB+11]. Também foram analisadas as VPLs, uma vez que podem ser uma boa solução para a programação por utilizadores finais. Em seguida foi feito um resumo das propriedades das VPLs e são dados exemplos da sua utilização.

Com o conhecimento adquirido criamos um protótipo para Android de forma a estudar como é que os utilizadores finais percecionam e aceitam uma VPL num smartphone. Neste protótipo os utilizadores finais podem ligar blocks e criar tarefas. Essas tarefas podem ser partilhadas e reusadas, e dependem de eventos que são gerados no smartphone. Com este protótipo tentamos fornecer respostas para questões como: *É possivel implementar uma VPL em um ecrã de pequenas dimensões? Qual o nivel necessário de abstração para não confundir os utilizadores finais e não limitar os programadores experientes? Como é que podemos integrar a reutilização de*

tarefas e de blocos sem confundir os utilizadores finais?

Nós fizemos muitas mudanças no nosso protótipo devido ao feedback de utilizadores finais, mas quando tentamos algumas melhorias sentimo-nos limitados pela linguagem Java. Para ultrapassar esta limitação, foram feitas algumas experiencias onde tentamos combinar Java com Prolog e Java com Scala. Essas experiencias permitiram-nos criar um sistema de regras de rescrita para reagrupar blocos automaticamente de acordo com um conjunto de regras definidas pelos programadores experientes.

Acknowledgments

I remember when this dissertation started. We were talking about an other dissertation, and suddenly we got an idea. That idea was developed in minutes, and immediately all potentialities were in front of me and my supervisor.

In the following months, we used some time to discuss some of the work that could be done. Together with Tiago these meetings were fast and engaging.

During the dissertation, other meetings followed, and they were essential to build the solution we have now. Discussing ideas and solutions was important, and I couldn't do it by myself. Therefore, I have to thank Professor Hugo Sereno Ferreira for getting some free time when I really needed, and Tiago Boldt Sousa, that could always find some time to give me fast feedback and accurate answers to my needs. Additional, I would like to thank Cristiana for always supporting me. Without them, my dissertation would surely be poorer.

This dissertation uses a template developed by Professor Hugo Sereno Ferreira in his PhD thesis. This template allows the reader to navigate easily in the document in the computer version, using bookmarks, pointers at the start of each chapter for each section and the feature I like most: pages where the references are being cited. If the reader clicks in a reference, then he can go back to the page he was since every reference has the pages where they are being cited. This template improved the value of my thesis and is one more reason to thank my supervisor.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

End-users are users that ultimately use software[1]. Most software users may be regarded as end-users, depending on the perspective: from the computer sciences student, to an end-user that only uses his personal computer to access his email. All programmers are end-users (to some extent), but all end-users are not programmers.

The quantity and complexity that end-users are increasingly demanding from their applications and devices makes it impractical for a software developer to 'foresee' every possible combination and explore every valid alternative.

Alan Kay shared the same concern in the early days of personal computers (PC), during the design of *Smalltalk* and *object-oriented programming* [Kay93]:

> *... there would be millions of personal machines and users, mostly outside of direct institutional control. Where would the applications and training come from? Why should we expect an applications programmer to anticipate the specific needs of a particular one of the millions of potential users? An extensional system seemed to be called for in which the end-users would do most of the tailoring (and even some of the direct constructions) of their tools* [*sic*].

Now, it is a time where smartphones are the technology that is expanding. According to Canalys [Can], in 2011, smartphones sales surpassed PCs [stPSi]. These portable devices are so useful that they are used every-day in several tasks [RZ10]. Smartphones have hardware with a

[1] The word "end" was attached to "user" by the economics and business to distinguish between a software buyer (an organization, per example) and a software user (an employer of that organization)

meaningful computation power, and they have sensors that can be used to build *context-aware applications* [SAW94].

Combining these two factors we have a set of tools that can be used to automate several daily tasks. But are those tools being explored to the maximum potential?

1.1 VISUAL END-USER PROGRAMMING FOR SMARTPHONES

Nowadays, there is a fair amount of expert programmers around the world. However, the amount of end-users is inevitably bigger. Scaffidi and Shaw estimated that there were 80 million end-user in America, in 2005 and, in 2012, there would be 90 million, contrasting with less than 3 million expert programmers for both years[SSM05]. As the number of end-users grows their needs become more specific, and it is impossible to have a professional programmer working working to fulfill every particular end-user need. A possible solution for this problem relies on empowering end-users with programming abilities.

End-user programming (EUP) was first referred by Nardi in a study about the use of spreadsheets in office workplaces [Nar93]. That concept spread within literature and different definitions appeared. EUP tries to create solutions to end-users, so they can do some sort of programming. With the increase use of EUP in the industry and the growing dependency around the artifacts generated by end-users a new area arose: *End-user Software Engineering* (EUSE) [BCR04]. EUP focus on the production of a program and EUSE is worried about transmitting concepts from software engineering into end-users, increasing the quality of the solutions created. Those concepts cannot be separated and they are deeply connected.

1.1.1 End-User Programming

EUP is referred in different contexts among literature. Some authors use this concept to refer to "novice" programming or "non-professional" programming and despite these users can share common problems with end-users programmers, we believe that those concepts are different. In a recent paper Ko et el. stated that the main difference between an end-user programmer and a professional programmer is the target client [KAB+11]. A professional programmer has an external client, while an end-user programmer programs for himself, being his own client. This difference has it source on the intent of the programmer and not on the level of knowledge in a programming language or in a programming ambient, as shown in figure 1.1 [KAB+11].

Under this interpretation, if an expert programmer decides to build a program for himself, then he is applying EUP. This label is dynamic and works as a vision from the application point of view. If the application had thoughts, it ought be ask: *Who made me? Who is using me?*. If the answer is the same person, then, the creator of the application used EUP.

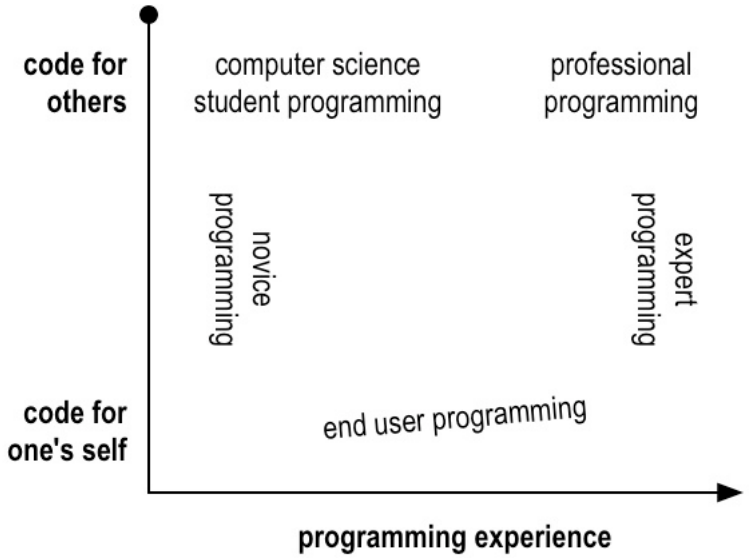


Figure 1.1: This diagram represents how programmers can be classified depending on their intents and experience. [KAB+11].

Although we believe that Ko et el.'s definition is a good attempt to define EUP. However, with this definition we can have cases like an expert programmer that is doing some scripting for himself. Therefore, we will narrow down that definition and use the following definition:

> *"EUP is the practice by which end users write computer programs to satisfy a specific need, but programming is not their primary job function." [LSBM09]*

Spreadsheets are a successfully case of EUP. Spreadsheets users grow each year, and complexity of the solutions generated by these users is increasing [Pano8]. EUSE defends that the increasing quality dependency in EUP solutions creates a necessity to empower end-users with software engineering concepts.

1.1.2 End-User Software Engineering

EUSE is an emergent research area that is concerned about quality in solutions developed by end-users programmers [BCR04]. EUSE focus on providing answers to concerns such as: : *How can we integrate testing and debugging within EUP development cycle? How can we facilitate and promote reuse?* Those questions have appeared recently in the literature with the necessity to provide tools to end-user programmers, so they can reduce, detect and locate faults. This necessity came after the observation in several studies that some of the programming made by end-users are very-important in organizations [Pano8]. A failure on EUP can cause impact on the organization [Seg07], and it is necessary to introduce some practices to prevent failure.

Teaching end-users the software engineering concepts is not the main focus of EUSE. In *The state of the art in end-user software engineering* [KAB+11] the authors stated:

> *"... the challenge of end-user software engineering research is to find ways to incorporate software engineering activities into users' existing work-flow, without requiring people to substantially change the nature of their work or their priorities..."*

1.1.3 Visual Programming Languages

End-users and novice programmers can be very efficient using *visual programming languages* (VPLs) [CK02, BH95, NPCn01, WNF06].

Research on VPLs [Gre95, KHA97, Bur01] usually defines this area as Myers defined in 1986 [Mye]:

> *"Visual Programming refers to any system that allows the user to specify a program in a two (or more) dimensional fashion. Conventional textual languages are not considered two dimensional since the compiler or interpreter processes it as a long, one-dimensional stream Visual Programming includes conventional flow-charts and graphical programming languages. It does not include systems that use conventional(linear) programming languages to define pictures"*

VPLs started with flowcharts, with the conviction that would be good for teaching novice programmers. Others approaches emerged such as: data-flow, logic, visual production systems, forms and object-oriented languages [Gre95].

This type of language tries to make easier to express and consequently, understand programs, using concepts as: simplicity, concreteness, explicitness and responsiveness [Bur01].

In 1995, Green [Gre95] stated what the VPLs could:

> *"(i) reduce the number of concepts needed to program (e.g. no variables), (ii) allow data objects to be explored directly, (iii) explicitly depict relationships, and (iv) give immediate visual feedback of updated computations during editing."*

We decided to do a prototype for smartphones since this platform is expanding and it lacks of VPLs solutions.

1.1.4 Smartphones

Shipments increased by 63% over the previous year, compared to 15% growth in PC shipments  [stPSi]. Smartphones' vendedors shipped almost 489 million smartphones in 2011, compared to 415 million PCs. Additionally, this devices already started to replace computers on some tasks [RZ10], and for some people are the only way to do them.

Smartphones came with a set of tools and services that provide an easy and fast road for application development. Furthermore, applications can spread almost instantly, since there is a market where developers can publish their work and make it available for all users.

When we refer to smartphones we are limiting our domain within the devices that have a screen size between 2 inches and 5 inches (measured diagonally). Screen size is a problem, mainly because human visual perception and attention is low on these sizes [CXF+03]. Processor speed is also a problem and, in general, smartphones warn when applications' computation exceeds some seconds [fR]. Battery's life dictates that every application should run the minimum time possible. Some smartphones have low memory available, and almost all smartphones have limited forms of input (the main method is the touch input).

Although there are some solutions to develop application for the smartphone using EUP and VPLs on a PC (per example, AppInventor [Smu11]), these devices still lack of good solutions that use EUP and VPLs to configure them without using a PC.

1.2 AN APPLICATION FOR EVERY TASK

Researchers in the EUP area already felt the need to start looking into smartphones. In a Google Tech Talk, Allen Cypher made an presentation about the evolution of the EUP [eoEUP] and finished it saying that mobile devices should be the next target. And we can understand why.

Sometimes users want simple applications to do ordinary tasks in their smartphone. Some of these tasks are sensor-dependent (or context-dependent [TOTK04, CK00]), and others are a simple routine. However, every time a new task is needed, usually it is necessary an application to do it.

If an end-user needs a simple task to be done automatically in his smarthphone he has two options: He learns how to program, or he gets a simple application for the task he wants. TouchDevelop [TMdHF11] tried to provide means so end-users could do some scripting for their smartphones. Although this is an interesting approach, we believe that the lack of a visual component can scare some end-users. There are other solutions like Tasker [Tas] and AutomateIt [Aut] that provide tools to create simple tasks. However, the set of possible combinations for those tasks is limited, and they miss basic EUSE concepts like reuse or debugging.

Even smartphones' developers need, sometimes, a fast and simple process able to schedule, automate, create and compose tasks. Everyday smartphone users do some tasks like turning the wifi on when they are on their home, or turning the 3G on, get emails, and then turning the 3G off, or even delete messages received by a number that is always spamming about events. Those tasks are examples that could be automated.

It is possible to imagine a framework that is flexible enough to cover most of the task combinations. That framework could be a hybrid platform for both end-users and expert programmers. Expert programmers could develop the pieces, and end-users could connect them. A set of pieces connected is a piece and pieces could be shared between end-users. This framework would simplify and speed up the task creation process.

We are developers, and mobile end-users, so it is hard to ignore the potentialities of such a framework. This challenge is so tempting that it is impossible to ignore it.

1.3 GOALS

Our work touches some areas that are hard to research. Therefore, our objectives are ambitious. We decided to provide a horizontal approach with ambitious objectives rather than a vertical approach in one objective, mainly because all areas are deeply connected.

We made a prototype to help us study:

- **Limitations of a smartphone in the implementation of a EUP solution**
  VPLs can be hard to use in a small screens. How can we use visual elements without overflowing the screen? Is a typical data-flow solution with zooming options enough? Can we use VPLs in small screens and work with solutions with medium to large number of components?

- **Defining an abstraction level for a collaborative framework**
  Higher abstraction levels may be good for end-users but they can limit the blocks produced by expert programmers. Can we achieve a balanced abstraction level for both end-users and expert programmers?

- **Impact of the reuse in EUP and contributions to EUSE**
  How can we approach reuse of components and incite end-users to use it? Could we benefit from a market for sharing components? What other reuse possibilities can be explored?

1.4 TECHNOLOGY

We chose Android among the operation systems that are available for smartphones. We made this choice because Android doesn't require a specific operation system to be developed[2], it is open source, and we already had some experience using it.

We used the Android Software Development Kit (SDK) in combination with the eclipse Integrated Development Environment (IDE). Android SDK uses the Java language combined with XML[3].

We also made some experiences using Prolog (§ 4.3 (p. 54)) language with an engine in Java. We've also researched some implementation variations by using Scala (§ 4.4 (p. 57)). We will explain the motive why we chose these languages and we will introduce them (along with the features that caught our attention) in Chapter 4 (p. 51).

[2] iPhone requires a MAC and Windows Phone 7.5 requires Windows
[3] XML is used to simplify the GUI creation

1.5 OUTLINE

In this chapter we introduced a set of concepts that are important to understand our thesis context. Next, in Chapter 2 (p. 9) we are going to go deeper and analyze the problems that are researched in EUP and EUSE. As we already stated, VPLs can be a good solution for EUP. Therefore, we are going to study this set of languages along with other languages that connected with VPLs: data-flow languages and visual data-flow programming languages.

In Chapter 3 (p. 25) our prototype will be presented, and it will be analyzed in comparison to the properties defined in Chapter 2 (p. 9). Then, in Chapter 4 (p. 51), we will try to improve our solution using Prolog and Scala. At last, we will summarize what we have done, and provide some guidelines to the future work in Chapter 5 (p. 67).

Typographical conventions were defined to help in the readability of this document. Concepts are typically introduced in *italic* and acronyms are usually in ALL-CAPS. Classes and parts of code are printed using mono-spaced type of font. In § 4.2.1 (p. 53) we defined a set of rules, and usually they appear with fixed-width characters. References and citations appear inside [square brackets] and in highlight color — when viewing this document in a computer, these will also act as *hyperlinks*.

Chapter 2

State of the Art

This chapter starts with an overview of the barriers that the research community found in EUP and EUSE. Then, we will take a look on some areas and applications that inspired our solution.

We found in the data-flow execution model a good common metaphor between end-users and developers, and we soon realized we needed concepts from VPLS to represent that metaphor.

In this chapter we will cover different subjects that are connected. We will do some abstraction and everything that is considered software or pieces of software will be called components.

2.1 END-USER PROGRAMMING AND SOFTWARE ENGINEERING

This section presents some problems that EUP and EUSE have that need to be solved. Additionally, we are going to analyze an example of a commercial solution that uses EUP in a smartphone.

2.1.1 End-User Programming

End-users generally lack experience on programming, and they want simple, direct and easy tools to achieve their objectives. Programming has challenges that are similar to both end-user programmers and to novice programmers.

Ko, Myers and Aung [KMA04] studied end-user programmers as they were learning Visual Basic.NET. They found out six learning barriers, and they associated each barrier with an end-

user thought. In the following list, there is a summary of those barriers. Each barrier has a thought quoted from the article, and we added a small description to each barrier.

- **Design** *"I don't know what I want the computer to do"*
  It is hard to think in a problem in an algorithmic way. End-users need to think in a way that can be translated to a computer understandable format.

- **Selection** *"I think I know what I want the computer to do, but I don't know what to use"*
  After the design barrier has been surpassed it is necessary to pass those thoughts to a readable format for computers. "What can I use?" it is usually the main question that end-users do when facing a selection barrier.

- **Coordination** *"I think I know what things to use, but I don't know how to make them work together"*
  All the components were gathered, but it still misses the links between them. The lack of knowledge about the (API) is part of this problem. End-users need to know the pre-conditions and pos-conditions of a component.

- **Use** *"I think I know what to use, but I don't know how to use it"*
  Use barriers result from the lack of knowledge or bad documentation about a component. Usually these questions are done in an use barrier: "What does this component do?" "How can I use it?" "What does it produces?"

- **Understanding** *"I thought I knew how to use this, but it didn't do what I expected"*
  End-users have expectations about the components they use. If those expectations don't match with reality (at compile or runtime), the end-user does not know what happened and where it happened.

- **Information** *"I think I know why it didn't do what I expected, but I don't know how to check"*
  The process of debugging and testing usually is something that end-users do not know how to do. Those processes come with programming experience and require knowledge of the program work-flow.

In 2006, Blackwell pointed out another problem in his work at programming psychology [Bla06]:

> *"... programming languages are universally designed by people who are themselves professional programmers... As a result, they eventually create new programming languages that they themselves would like to use. The design of a language for use by end-user developers cannot rely on such intuitions, because no language designers are themselves end-users."*

This problem limits EUP solutions because developers usually forget that end-users do not share the same knowledge representations as expert programmers.

2.1.2 End-User Software Engineering

EUP started to be used in complex and critical solutions. An example of the importance that EUP has are the spreadsheets. Spreadsheets are still growing and the common errors that are made in this type of EUP were already studied [Pano8]. The impact of some errors were felt in business [TT97] and organizations were created to track costly errors in spreadsheets to alert end-users, like European Spreadsheet Risks Interest Group [Eur].

After realizing the importance of EUP it was necessary to start a research that could empower end-users with software engineering concepts on their applications. Therefore, EUSE is interested in five concepts [KAB+11]:

- **Requirements**
  In EUP, the requirements come from the end-user. Therefore, requirements are implicit and easy to understand, but are more likely to change. Requirements are discovered through experimentation. End-users try the tools and they see what they can do and requirements emerge from those experiences.

- **Design and Design Specifications**
  Since the requirements are always changing, the solution design can't be detached from the requirements. Requirements and design usually are made together in EUP. It is common that end-users can only understand the constraints on their solution just when they are programming. In addition, the value of a good design can only be felt at long term. An example of a property that can only be valued in long term is scalability. Good solutions are also more valued if they need to be shared with others. Since end-users usually work alone a pleasant design specification is hard to value.

- **Reuse**
  It is easier for an end-user to reuse than start something from scratch. Reuse is useful to save time on the development of one solution, and to start using a tool[1]. However, finding the artifact to reuse is hard and, usually, this problem results in a selection barrier. Commonly, end-users need experienced peers to have an example code [KMA04]. After a usable component is found, it may be difficult to use it (*use barrier*) or to connect it to the solution (*coordination barrier*). Although users can create components for future reuse, examples of a specific case are the most used form of reuse [KMA04].

- **Testing and Verification**
  One of the problems with end-users is overconfidence. On studies about end-users and spreadsheets, end-users showed a high confidence about the correctness of their solutions

[1] Copy-pasting code is one example of reuse that is commonly used when programmers start to learn a new language.

[Pano8]. This overconfidence prevents end-users from testing and thinking in the robustness of their solution.

- **Debugging**
  After proving that an error exists through testing, it is necessary to locate and fix those errors. Facing a situation where debugging is needed can lead to an information barrier. Debugging may require a set of tools like print statements or breakpoints. It is necessary to know where to use those tools and what to look for. Since most of the end-users do not understand the execution of their programs it is hard to create a hypothesis about a program failure. Another problem is the end-user priorities. End-users prefer a program that works than a program that doesn't fail[2]. Therefore, sometimes, instead of fixing the problem, they add some code, or they modify what they have done, so it only works in the particular case where it failed.

2.1.3 An Example of EUP on a Smartphone: Tasker

Tasker [Tas] is an application developed for Android to empower end-users with the ability to define tasks depending on a context. We imagine this application as a subset of our application, in the future, so we decided to analyze it. We want to know if we would be able to do, at least, all that Tasker does, with a different level of abstraction.

Concepts

Tasker defines a set of concepts in their application and documentation [Tas].

- **Context**
  It is a trigger that will initiate the tasks. A context can depend on time, date, location, state, when an event occurs or when an application is running.

- **Action**
  It is something that the smartphone can do, like sending a SMS or activate silent mode. Tasker has currently more than 190 built-in actions available in 14 categories. An action has parameters that need to be filled and can have conditions (`if` clauses) that can be used to dictate if an action will be triggered or not.

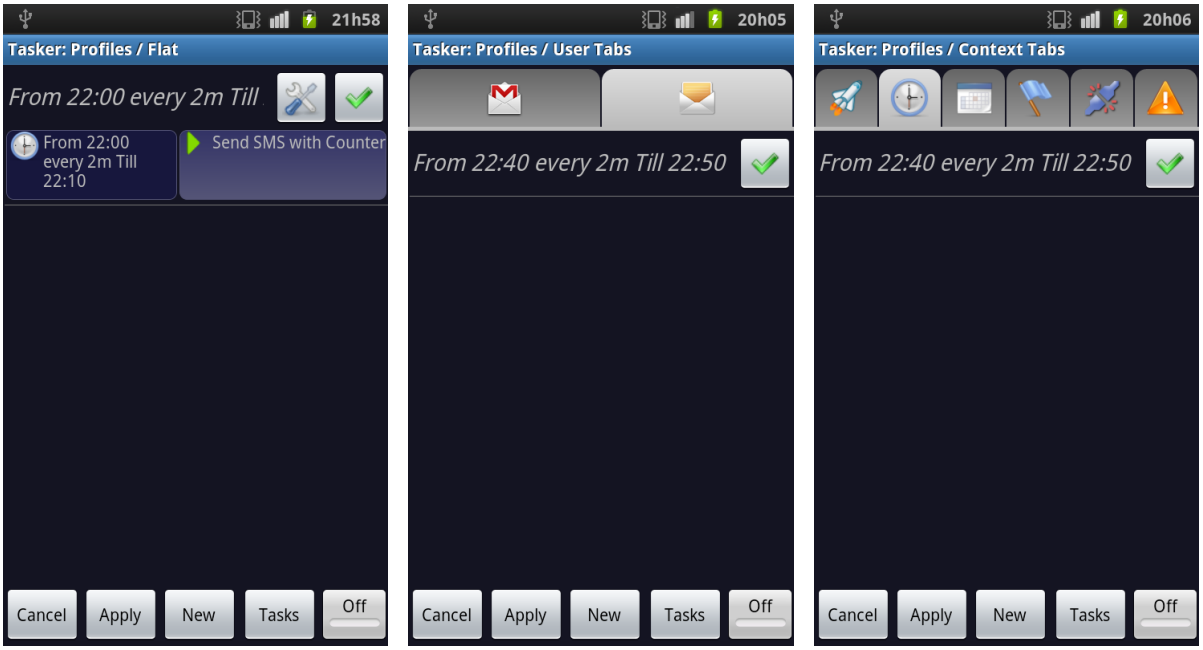
- **Task**
  It is a set of actions. It is not guaranteed that the actions will be run in the order they are specified.

- **Profile**
  It is combination of a context with a task.

[2] Although for some programmers a program that works is a program that doesn't fail

In Tasker we can create profiles that do a task that depends on a context. Each task has a set of actions and can be reused (but not shared).

Functionalities Description



(a) Flat View (b) Costum View (c) Context View

Figure 2.1: Initial menu in Tasker with a profile that sends a SMS every two minutes with a variable from 22:00 to 22:10. Profiles can be viewed in three modes: Flat, Custom and Context

The first menu of Tasker shows profiles that were defined. This initial menu has three forms of view: a flat view (Figure 2.1(a)), where all profiles are seen; a context-based view (Figure 2.1(c)), with tabs that group profiles by their contexts; and a custom-based view(Figure 2.1(b)), where the user can create tabs and arrange the profiles the way he wants.

Touching the new button at the initial menu like the one in Figure 2.1 opens a pop-up to define the profile name. Then, we need to select a context from a category showed in Figure 2.2(a) (p. 14) so we can select a task already created or define a new one. A dialog similar to the one in Figure 2.2(c) (p. 14) but with no actions appear. After pressing the + button it is possible to select an action from the action categories list, similar to the one in Figure 2.2(b) (p. 14). An action needs some parametrization and can have a set of if clauses. Actions can use a set of pre-defined variables, such as %WIFI that has the current wifi status (on or off). Defining new variables is also possible, using the action "Variable Set" to initialize them. In Figure 2.2(b) (p. 14) the variable %COUNT was not initialized, therefore the first message will print %COUNT and not the expected number. Tasker also allows loops in actions with the action GOTO.

A task can be reused after its creation. It is also possible to export a task or a complete profile. This feature is useful to backup existing tasks/profiles.

(a) Contexts categories (b) Action categories (c) Task composed by actions

Figure 2.2: First image shows the available context categories. Second image shows the available actions categories and the last image shows how actions are displayed inside a task.

Critical Analysis

Tasker showed us that is possible to automate some tasks. However, it lacks of a visual language with reusable components. Only a task (set of actions) can be reused, but they can't be connected with other tasks. Also, reusing is not customizable since there are no inputs or outputs.

Tasker allows a mode that saves debug information on a file, so end-users can't see debugging information in real time.

2.2 THE DATA-FLOW EXECUTION MODEL

A data flow language can be defined as follows:

> *"... any applicative language based entirely upon the notion of data flowing from one function entity to another ..."[DK82]*

The data-flow execution model appeared in the computer research as a form to do parallel computing [Ack79, JHM04]. This model is based in the *availability* of the operands. If an input is ready, the operand is applied. This provides a valuable property for parallel systems: asynchronous execution.

Another advantage of this execution model is that program definitions can be represented graphically [DK82, Ack79].

2.2.1 Data-Flow Graphs

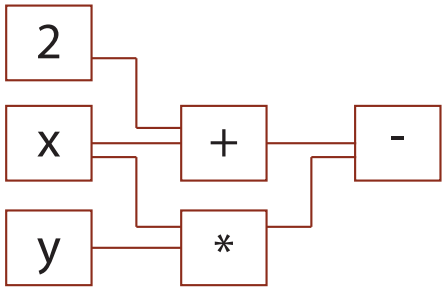


Figure 2.3: Data-flow graph example for operations: A:=2+x; B:=x×y; C:=A-B

A data-flow model can be illustrated using a data-flow graph, where nodes can be tokens or functions (Figure 2.3). When a node's input is available, the node can absorb it and produce an output. Generally, this kind of model is compared to a factory, where machines are nodes that receive some sort of input and can produce an output. In Figure 2.3 the number 2, and variables x and y are ready, so they are fired. The tokens + and * receive them and can be computed in parallel. Then both produce an output that is consumed by the token -.

This kind of representation makes it easier to evaluate a program, and it was already used in teaching novice programmers how to program [AA92].

However, the graph doesn't need to be executed from the left to the right. In fact, data-flow model has two approaches for its execution [JHM04]:

- **Data driven** In this approach a node only executes when he receives new data.
- **Demand driven** This approach visits the graph from the end to the start. Each node request data from the nodes attached.

2.2.2 Properties

Ackerman, in 1982, defined five rules that are needed in a data-flow language [Ack79]. Lee made a good summary of three of those rules [LH93] that we will use next:

- **Freedom from side-effects**
  "This property is necessary to ensure that the data dependencies are consistent with the sequencing constraints. Dataflow model of execution imposes a strict restriction by prohibiting any variables from being modified. This is done by using "call-by-value" rather than "call-by-reference" scheme. A call-by-value procedure copies rather than modifies its argument which avoids side-effects."
- **Single assignment rule**
  "This offers a method to promote parallelism in a program. The rule prohibits the use of

the same variable name more than once on the left-hand side of any statement. The single assignment rule offers both clarity and ease of verification, which generally outweigh the convenience of reusing the same name."

- **Locality of effect**
  "This means that instructions do not have unnecessary far-reaching data dependencies. This is achieved by assigning every variable a definite scope or region of the program in which it is active and carefully restricting the entry to and exit from the blocks that constitute scopes."

- **Data dependencies equivalent to scheduling**
  A data-flow program execution depends on the data dependency between instructions. There isn't an explicit sequence of instructions because instructions will run when their data is ready. The dependency that is created is similar to scheduling. It is only possible to execute a task when the resources for that task are available.

- **Lack of history sensitivity in procedures**
  There is no state variables between instructions in a data-flow execution model because a node only receives inputs. There isn't a global storage unit.

Data-flow languages had some problems to define the grain for each node [JHM04]. If data-flow uses fire-grained nodes like shown in Figure 2.3 (p. 15) then the graph would easly get a complex visual representation. Another problem that pure data-flow languages faced was the lack of good solutions for iteration and conditional constructs. This problem was later solved as seen in § 2.4 (p. 19).

The data-flow graphs inspired VPLs, and some of those languages have an execution model inspired in the data-flow execution model.

2.3 VISUAL PROGRAMMING LANGUAGES

Since the early days humans used images as a means to communicate and even now, in software engineering, we keep using it. An example of a visual description of a problem instead of a textual one is the Unified Modeling Language (UML) [OMG10]. UML provides a set of visual tools to represent solutions, and it is used world-wide by expert programmers. VPLs try to combine the communication power of a visual language with the possibilities of a programming language.

Advantages of VPL are hard to generalize, because they depend on the context they are applied, and they also depend on the target end-user. Baroth, on his experiments on VPLs usage on the production of measurement systems [BH95], stated:

"The advantages/disadvantages of any programming environment are dependent on the context in which they are being used."

Some research has been made on advantages and disadvantages of VPLs and showed that end-users and novice programmers can be more efficient using VPLs [CK02, BH95, NPCn01, WNF06]. In general, it is possible to conclude that VPLs are useful because:

- require fewer concepts to program [BBB+95]
- optimize access to semantic information [NPCn01]
- facilitate the program comprehension [NPCn01]
- are less error prone [CK02, WNF06]

However, these languages also can also have problems such as [AM90]:

- need for an automatic layout
- handling large programs or large data [Bur01, BBB+95]
- difficulty in building editors and environments
- lack of formal specification
- lack of portability options[3]

Despite some VPLs can solve some of the problems above, all these languages usually share some characteristics.

2.3.1 VPL's Characteristics

VPLs have a set of common approaches [Bur01, BBB+95]. Those approaches, or strategies, create a set of characteristics that are important in this kind of languages:

- **Concreteness**
  Some languages are powerful because of abstraction. However, concreteness goes in the opposite direction. VPLs tend to be concrete, using real values, contrasting with types of values.
- **Directness**
  Instead of working with possible values or objects the user works directly with a concrete value or object.

[3] Textual languages are easy to share and can be read anywhere. VPLs, on the other hand, require specific programs to be displayed.

- **Explicitness**
  In textual languages, the relation between objects is implicit. In VPLs the more explicit those relations are, the better.

- **Immediate Visual Feedback**
  Changes in a VPL should be seen and felt immediately, without any type of compilation.

Although VPLs share some properties, they can be classified into different categories.

2.3.2 Classification scheme

In 1994, with the increasing research on VPLs, Burnett suggested a classification for subsets of VPLs [Bur94]. However, this classification was too specific. Boshernitsan et el. launched, in 2004, a summary of the visual programming types, classifying them in: Purely Visual Languages, Hybrid Text and Visual Systems, Programming-by-example systems, Constrained-oriented systems and Form-based systems [BD04].

Although Boshernitsan's summary [BD04] is useful to characterize several systems, we believe that the name given at programming-by-example can be misleading. According to Brad A. and Myers, example-based programming has two forms: programming-by-example and programming-with-example [AM90]. Therefore, we will change the name of programming-by-example on Bosthernitsan's summary to example-based programming and we will divide it in: programming-by-example and programming-with-example.

- **Purely Visual Languages**
  This VPL is compiled directly from the visual form. Debugging and programming are both made in a visual environment.

- **Hybrid Text and Visual Systems**
  The visual form of this system is a layer that is translated to a text form. In this kind of systems, it is possible to program in a text or a visual form and alternate between visualization modes.

- **Example-Based systems**
  Example-based-systems can be divided in: Programming-by-example and programming-with example. **Programming-by-example systems**, or "automatic programming", *"tries to guess or infer the program from examples of input or output or sample traces of execution"* [AM90]. **Programming-with-example systems** *"require the programmer to specify everything about the program (there is no inferencing involved), but the programmer can work out the program on a specific example. The system executes the programmer's commands normally but remembers them for later re-use"* [AM90]. This kind of systems can be seen as a macro-building systems.

- **Constrained-oriented systems**
  In this kind of systems, there is a set of constrains that are built to create rules at a certain environment. This technique is useful to construct simulations, dynamic documents, and manipulable geometric objects [Bor86].

- **Form-based systems**
  The most known example of this system are spreadsheets. This system is characterized by the presence of cells that have connection between them.

Text input in smartphones is less efficient than on a PC, so we didn't consider hybrid systems. Example-based systems systems are useful to reproduce specific tasks but they lack of means to create triggers that schedule those tasks. Constrained-oriented systems are interesting for end-users who have a mathematical/logical knowledge. Form-based systems provide a metaphor of cells with a single connection limiting the solution possibilities.

Although an constrained oriented approach could result, we decided to focus on a purely visual language that could be easily understood by almost all end-users. In this dissertation, we will focus on one type of those systems: visual data-flow programming.

2.4 VISUAL DATA-FLOW PROGRAMMING LANGUAGES

Data-flow has a good metaphor that can be used in a VPL. The idea of data flowing while it is filtered to produce new data is an inspiration of many known VPLs such as LabView [San90] or Prograph [CGP89]. They share some properties from this model and usually they have some structures that try to compensate the lack of iteration and conditional constructs in the execution data-flow model.

This subset of VPLs is called *Visual Data-Flow Programming Languages* (VDFPL) and it gathers interesting properties.

2.4.1 Properties

Marttila-Kontio made a good summary about the VDFPL characteristics [MKRT09]. Some were already seen in § 2.2 (p. 14) but we decided to sum up all properties so we can see the common aspects that are shared with the data-flow execution model. We converted the Marttila-Kontio's summary into a list so we can have a enumeration of properties that can be used to clarify if a VPL is a VDFL.

1. "A visual data flow program code is presented as a directed graph."

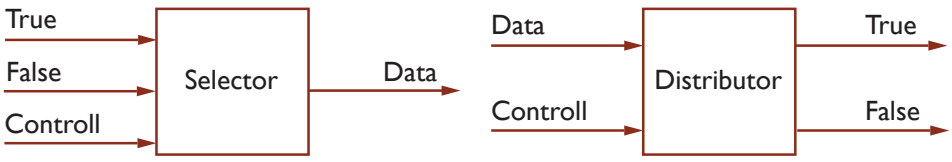
2. "A node represents, for example, a simple computational unit or a subprogram consisting of other nodes. Also, a node can represent systems input or output value."

3. "A node can have zero or more input and output arcs."
4. "An arc can be considered as a variable transferring data tokens from a node to another node or nodes."
5. "The arc is attached into the node through data ports or terminals."
6. "The data type of an arc, and a token it transfers, has to be the same type as the input and output terminals it is attached to."
7. "... a node becomes executable as soon as it has received all its input data and its (possible) output arcs are empty."
8. "An arc transferring tokens into a node becomes empty immediately when a node has absorbed a token or tokens from it."
9. "If a node does not have any inputs, it is immediately executed when the program is executed. Also, a node without any inputs is executed only once during the program execution."
10. "Once a node has been executed, it produces new values for all of its outputs."

Despite VDFPLs have this common properties, conditional constructs and iteration constructs are usually solved differently in each VDFPL. We will try to cover most of the cases presenting some of the inspirations that VDFPLs use.

Conditional Constructs

The solution to the condition constructs in the data-flow execution model exists almost as much time as the model itself. Davis and Keller [DK82] pointed out the solution to this problem through the use of a *selector node* (or *merge node*) and a *distributor node* (or *swith node*).



(a) Selector node (b) Distributor node

Figure 2.4: 2.4(a) shows a selector node that selects one input depending on the control value. 2.4(b) shows a distributor node that sends the data throw an output depending on the control value.

A selector node (Figure 2.4(a)) has three data inputs and one output. Data inputs are: a boolean control input and two data inputs called "True" and "False". Depending on the control value the respective data input will be selected.

A distributor node (Figure 2.4(b) (p. 20)) has two outputs and two inputs. One input is the control input, similar to the selector node. The other input is a data input. Depending on the control value the data is redirected for the "True" output or the "False" output.

Iteration Constructs

Iteration constructs have many solutions in the VDFPLs [MP00]. In one of the most recent surveys on this area [JHM04] the author classifies iteration as an "open issue" in the data-flow area.

Despite the differences among solutions, usually this problem is solved with an output from a node connecting to an input of a node that was already processed, creating a cyclic graph (Figure 2.5. If the nodes used in the loop aren't internally ready to stop the loop, they can be ended using conditional constructs.

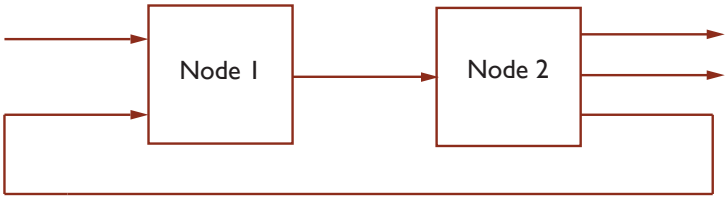


Figure 2.5: An example of an iteration construct that can be used in VDFPL

2.4.2 A VDFP Example: Blender Composite Nodes

Composite Nodes is a type of view in Blender. This type of view empowers end-users with a tool that allows them to build a visual script that can be applied in their scenes. The wiki from blender [Wik] gives an insight into the features that are provided in this tool.

> *"Compositing Nodes allow you to assemble and enhance an image (or movie) at the same time. Using composition nodes, you can glue two pieces of footage together and colorize the whole sequence all at once. You can enhance the colors of a single image or an entire movie clip in a static manner or in a dynamic way that changes over time (as the clip progresses). In this way, you use composition nodes to both assemble video clips together, and enhance them."*

An example of the representation used can be seen in Figure 2.6 (p. 22). The viewer node is a node used to print the output on the screen in a panel, that is not shown on the figure.

Concepts

There are three main concepts defined in Composite Nodes: Nodes, Noodles and Node Groups.

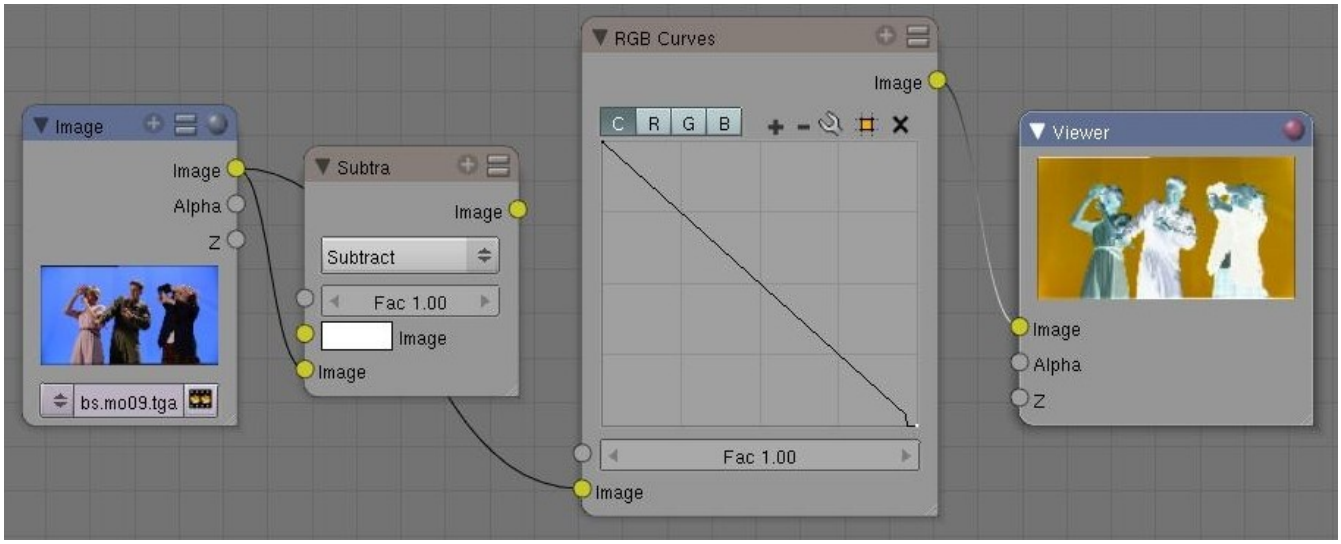


Figure 2.6: An image being inverted using Blender Composite Nodes [iBuNabi].

- **Nodes** A node can be seen as a function that can take inputs and can produce a set of outputs. There are three types of nodes: *Input Nodes*, *Processing Nodes* and *Output Nodes* [Wik]. **Input Nodes** exist to produce information. An integer value is an example of a *Input Node*. **Processing Nodes** can apply filters or transformations on their inputs to produce a set of outputs. Filters like blur or contrast are examples of a processing node. **Output Nodes** are useful to finalize a composition and specify were the result will be saved. A viewer node to display the output or a file output node are examples of those kinds of nodes.

- **Noodles** A node has configurable parameters, and its outputs can be connected with the inputs of other nodes, creating a network called *Noodle*.

- **Node Groups** Nodes can be grouped into a single node creating a *Node Group* that can be connected with other nodes.

Functionalities Description

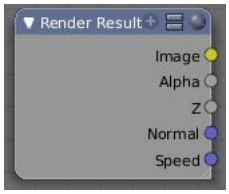


Figure 2.7: Possible colors on the sockets of a node [Web].

Output sockets should connect with input nodes of the same type. Therefore, to identify sockets types Blender Composite Nodes has a color for each type (Figure 2.7). There are three

types available: RGBA - yellow; three-dimensional vector - blue; value - gray. When different socket types are connected a default conversion will occur as follows:

- **Vector to Value** - Average (X+Y+Z)/3.0
- **Color to Value** - BW average (0.35*R + 0.45*G + 0.2*B)
- **Value to Vector or Color** - Copies value to each channel
- **Color to Vector** - Copies RGB to XYZ
- **Vector to Color** - Copies XYZ to RGB and sets A to 1.0

Blender Node Composition execution is optimized so every editing event (like changing a Node parameters) can trigger another execution. Optimizations like multi-threading usage are important, but one step in the optimization that is quite relevant is the dependency check. The Blender Node system sorts nodes based on their dependency. This way only nodes that depend on nodes that got tagged with changes are going to be updated.

Critical Analysis

It is important that the end-user can see how nodes can be connected just looking at the colors. Combinations between different types can be useful but can also be dangerous. Without information, end-user can fall in a *coordination barrier* or *understanding barrier* (§ 2.1.1 (p. 9)).

Another important thing is the ability to group nodes. This feature can be a good approach to solve the scale-up problem associated with VPLs [BBB+95].

2.5 CONCLUSIONS

There is some research in VPLs, EUP and EUSE, but each problem usually results in his own solution. We want to focus on some problems of EUP and EUSE. Those problems are: selection, coordination and information barriers from EUP; Reuse and debugging from EUSE. VPLs offers some properties that could help in the solution of these issues. *Explicitness* can help to surpass the coordination barrier, *directness* and *concreteness* can aid in transcending the selection barrier, and *immediate visual feedback* can be used to do debugging and to beat the *information barrier*.

There are two applications that can inspire our solution: Tasker and Blender composite nodes. Tasker provides a set of concepts but also tell us that what we want to do is possible. We see Tasker as a subset of our application. We want to provide tools so end-users can have an application that does all that Tasker does and more. Blender composite nodes are a visual inspiration. When we saw this application we thought that this fits perfectly on the concepts that we were imagining.

Chapter 3

Case Study

VPLs are useful to solve some problems in EUP and EUSE. Therefore, we are going to use some properties of these languages in our prototype.

The main influence on this work was one application that uses a VPL called: Blender Composite Nodes (§ 2.4.2 (p. 21)). Since this solution was made for PCs and we are working with smartphones, some changes were needed.

3.1 WHAT IS NEEDED?

We wanted to empower end-users with the ability to define tasks based on events. However, the task creation should be based in the connection of components. Each component should be able to:

- Receive information from other components

- Produce data for other components
- Connect to other components
- Be developed by an expert programmer
- Be shared

If a component had this set of properties the end-users could ask for new components, get new components, and share connected components. However, this idea is too abstract so it was needed something to represent it.

3.2 OVERVIEW

A set of concepts were defined to transmit ideas for end-users. However, most of those concepts are based in a single idea: *block*.

3.2.1 A Block Abstraction

A block is an abstraction that was already used in the software engineering. One example of that usage is black box testing [MSB11]. The idea behind black box testing is: We want to test a functionality, and we don't want to understand what the program is doing. We only know that giving a specific set of inputs, we should have a correspondent collection of outputs.

Blocks are also used in some VPLs. In fact, D. Hills used the term "box-line representation" to characterize some of VPLs in his survey [Hil92].

Block Based System

Zin defined a set of conditions that a block based system for EUP should have [Zin11]. However, he did not implement and tested a solution.

Zin described the block-based programming system as follows:

- "It should support software development in many problem domain"
- "Many blocks will be made available for each problem domain"
- "Each block supports a certain task or function"
- "End-users are allowed to customize blocks and to build applications adapting to their needs"
- "Application software development can be done by integrating these blocks"

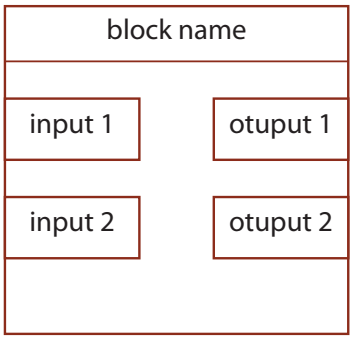


Figure 3.1: A block has a name and can have several inputs and outputs.

Zin also said that the blocks should be developed by expert programmers, converging with our ideas.

For us, a block has both inputs and outputs and it can be represented like shown in Figure 3.1. Each block can be seen as a function that absorbs inputs and produces outputs, like a data-flow node. However, in our case, we realized that we needed some block types.

Events and actions need a representation in our application, and robotics has some terms that were useful to name what it was needed in our problem domain: sensors and actuators. Sensors receive events and actuators perform actions.

Sensors

A *sensor* block usually has only outputs. A sensor provides information that all other blocks can use, and it is not activated by its inputs availability, contradicting one of the properties of the data-flow execution model. We could provide a boolean input in each sensor and each time a new boolean arrived that sensor block would activate. However, we thought that this approach would confuse end-users because they could think it was necessary to connect the boolean input with other blocks.

Some examples of sensors are: A time sensor that provides the current time or an orientation sensor that changes when the smartphones' orientation changes.

Actuators

An *actuator* is a block that executes some action on the smartphone, and usually it has only inputs. Contrarily to sensors, an actuator will almost always be a non-functional node. A sensor receives information and updates their outputs according to that information. Actuators, on the other hand, receive information to customize its behavior, and usually they do some change on the smartphone state.

Some examples of actuators are: An actuator that sends a acronymSMS with a content to a number provided by its inputs or an actuator that repeats an alarm with a configurable interval.

Input Providers

An input provider is used to customize inputs. They are useful for cases that the end-user wants a specific input. Imagine, for instance, that an end-user wants to use the send SMS actuator. Using an input provider, he can configure both the number and the text to be send.

Usually input providers have only outputs and each output is directly configurable. In fact, the first time an end-user tries to connect an output from an input provider, it will be prompt an interface so the end-user can set the value of the output.

3.2.2 Connectors

A *connector* can be an input or output. An output can connect to several inputs, but each input can't be connected to more than one output.

Each connector have a type and a value. Blocks are responsible for the output values while the system is responsible for the input values. Blocks set their outputs and the system propagate those outputs to the connected inputs.

A connector type is variable, and it can be anything. One connector can have a type that is an object created from the class `Car` or a built-in `Integer`. To connect an output with an input, it is necessary the same type.

3.2.3 Connections

Each block only executes when all data it requires is available. However, since we have sensors that can change their status at any time, providing new data, we can lose information.

We can conceive a block with an Input A connected to a time sensor that changes every minute. If the time changes from 15:00 to 15:01 our block has the value 1 on input A. However, our block needs more inputs that aren't available yet, so the time keeps changing and A keeps changing. What if we want to use the first value? Or even all values? Should we record all changes? The answers we found for those questions were connections.

We implemented the connection concept and we test it, but the user interface for this part was delayed every time. This happened because we couldn't find a way to easily configure connections in the user interface.

A connection can have three types:

- **Last Value**
  Last value connections behave as normal connections in a data-flow execution model. They have no memory and they only hold one value.

- **FIFO (*First In First Out*)**
  FIFO connections can save multiple values and return them by the order they were put in the connection.

- **LIFO (*Last In First Out*)**
  LIFO connections also can save multiples values, but the first values that are returned are the most recent.

A block could request all info or just a subset of the info to FIFO or LIFO connections.

3.2.4 Tasks

A set of connected blocks is a task, and a task is a block. When a task is created a new block is added to the list of usable blocks, and can be connected with other blocks. A task can be shared among end-users and can be disabled or enabled. If a task is disabled its blocks will not receive events.

3.3 ARCHITECTURE AND IMPLEMENTATION DETAILS

Our architecture has three important modules:

- **Core**
  This module provides a framework that can be used in all platforms that run Java. This framework was thought so it could be independent from the smartphone, and it has all abstractions necessary to build data-flows using blocks.

- **Android Specific**
  We needed a small layer for our problem context. This layer also introduces the concept of Input Provider. We decided to take Input Provider out of the core because it depends too much on the platform that uses the core module. This layer also creates new interface for Sensor and Actuators to facilitate block creations in Android for developers.

- **Android GUI**
  This module is responsible for the end-user interaction with Android. It also has some blocks implemented.

In Figure 3.2 (p. 30) we show what are the classes we defined in the core module and in the Android specific module. Almost all classes have a transparent name that can be related with the concepts we introduced in § 3.2 (p. 26). However, there is a class that doesn't have a direct correspondence with the concepts we introduced: *Group Block*.

We didn't want to enforce the task concept because a task in a smartphone can be different on a computer. So we put the task out of the core, but we didn't place it in the android specific layer, so our framework could be minimalist [Fra]. Therefore, only the core module plus the android specific module result are part of the framework that developers can use. However, a

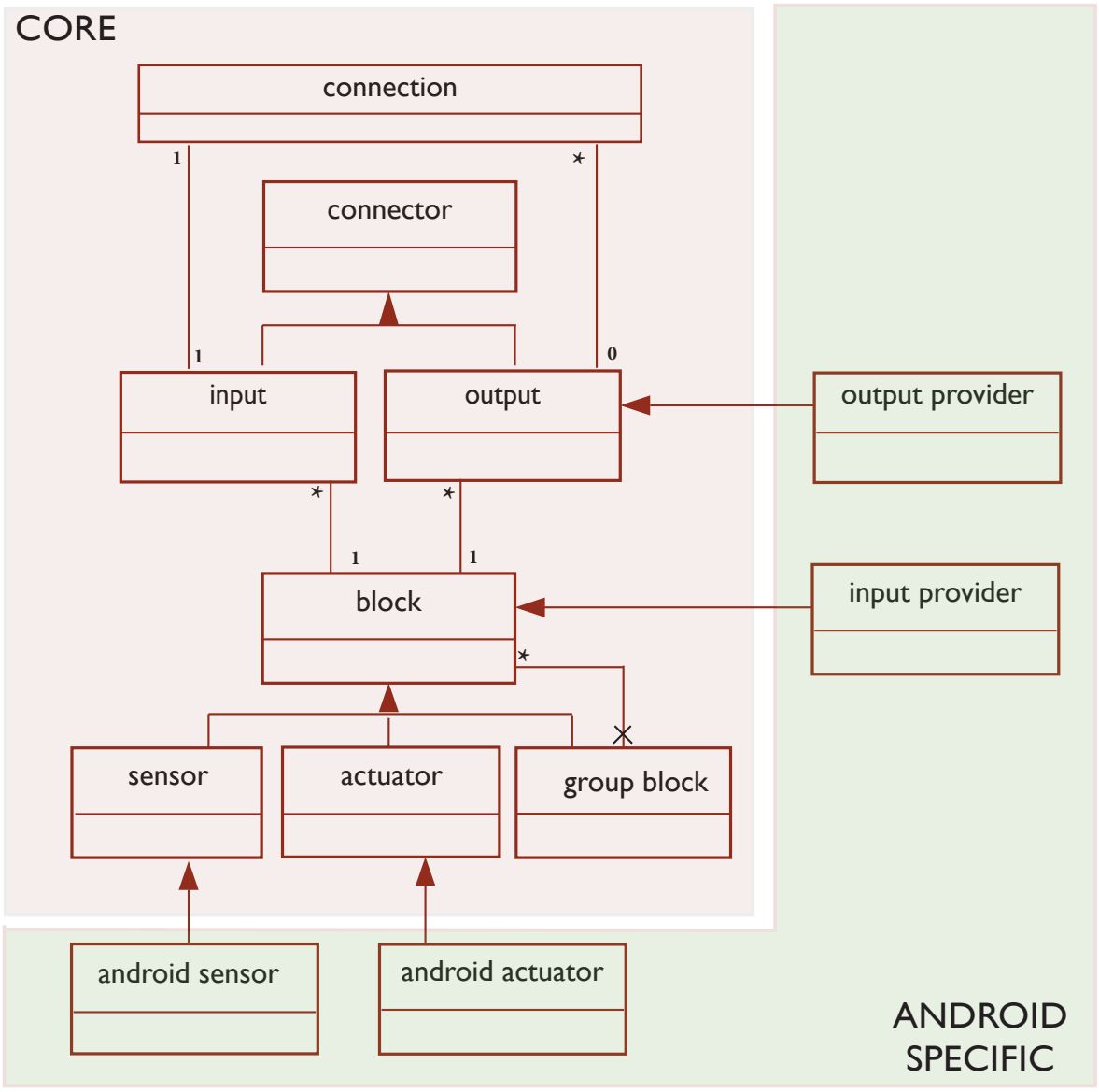


Figure 3.2: This UML shows the classes created in the core and the layer that is specific in Android.

task generates a block, and we couldn't use a standard block to represent the generated block. We needed to create a new class that could hold the information of several blocks. We called that class: Group Block. This class is used only to visualization proposes. As shown in Figure 3.3 (p. 31) group blocks don't have connectors on their own, but they have a reference to internal connectors.

Group blocks are created automatically when a task is created. We look to the task and check all its connectors that can connect with the "outside" and we create a block that points to those connectors.

If an input provider wants to supply a variable input it has to use an output provider. Output providers have a specific interface that enforces developers to develop a way to introduce the data of their kind of output. This is necessary because we don't know what kind of data output has,

since outputs can hold objects defined by developers.

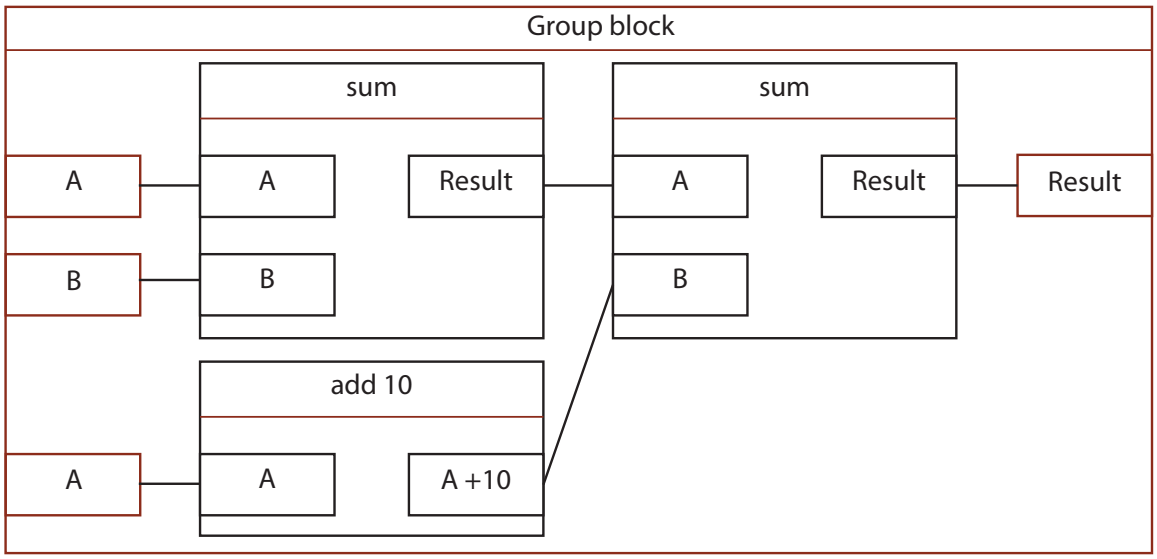


Figure 3.3: Group Block representation. The connectors of a group block already exist, and the group block only points to them.

3.3.1 Events, Dynamic Class Loading and Tasks Serialization

When the application starts a *service*[1] is created and it can be destroyed whenever the end-user wants, by pressing a button at the top bar in the initial menu. This service registers and receives all *broadcasts*, that are needed by the sensor blocks. A broadcast, in Android, is a message emitted by the system to all applications. When a broadcast is received, the service searches for the sensors that registered that broadcast, and delivers the broadcasts' intent [2].

Another two actions happen when the application starts: dynamic class loading and tasks loading.

Dynamic Class Loading

We didn't do a platform that could show how blocks and tasks could be shared. However, we prepared the basic mechanisms for that. Dynamic class loading uses a `ClassLoader` from Java to load a class only using its name. We have a set of names, we load them all, we store the respective class files in an `ArrayList`, and those class files can be used to construct objects.

If need to add a new block type, we add a line in the Java code like shown in Source 3.1 (p. 32).

[1] A service is a component provided by the Android operating system that allows some long term processing in the background [Ser]

[2] Broadcasts provide extra information in their Intents. An intent is usually related with an operation. A SMS intent can have, for instance, the number that send it and the content [Int]

```
1 actuatorClasses.add(classLoader.loadClass(classPackage + "SendSMS"));
```

Source 3.1: A line of code needed to add a block.

Where classPackage has the value "pt.tiago.thesis.android.actuators". We made the loading this way, so it could be easy to iterate over all files in a folder. That folder could have the name of the type of the block (sensor, input provider or actuator) and we could load the respective classes.

Tasks Serialization

A task has to be serializable so it can be shared. Therefore, we had to make sure all classes used by a task were serializable. However, we marked some fields inside each class as not serializable so they were cleared. When a task is shared from one end-user to another some information needs to be reset, that information is: logs from a block and values from connectors. Logs are cleared because the new end-user doesn't need them. Connectors' values are not preserved to prevent tasks that only depend on input providers to be executed when they are loaded. Imagine a task with an input provider that has a value, and a block connected to this task. This task would run each time we loaded it.

We didn't make a platform where end-users could share tasks. We tested the tasks' serialization saving each task created in an application usage. We preserve tasks every time we exit the application, and we load all tasks when the application starts.

3.4 A TOUR ON THE PROTOTYPE

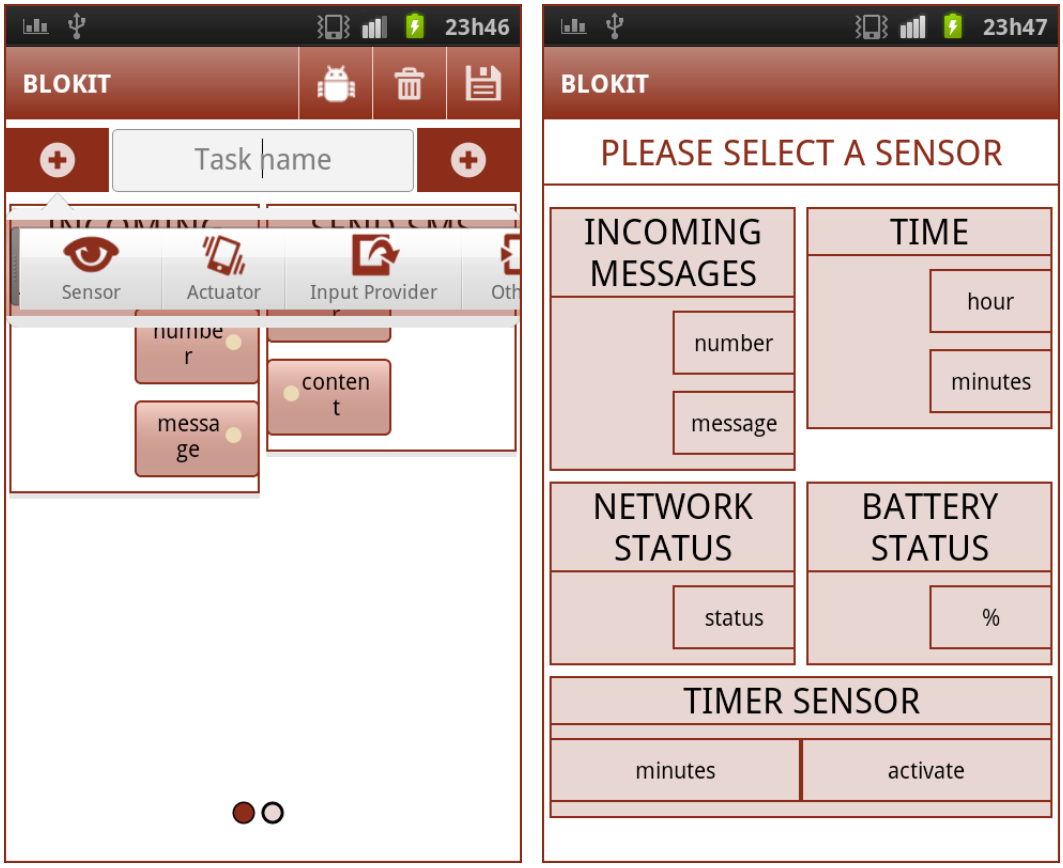
We already saw some of the concepts we defined for our prototype and how they are connected. However, we didn't show how those concepts pass to the user and how they are represented. In this section, we want to show how we represented the concepts defined in § 3.2 (p. 26).

3.4.1 Creating a task

To enter in the task creation *activity*[3] the user has a button at the top that appears in the main activity (or the main menu). In this activity we have two columns. Each column is a scrollable list that can have several blocks. Blocks from the left connect to blocks from the right. Therefore, blocks from the left only have buttons at each output and blocks from the right only have buttons at each input.

We have two buttons to add blocks. One on the left and other oh the right. In Figure 3.4(a) (p. 33) we can see what appears when the left button is touched. This kind of bar is

[3] An activity is a screen with a user interface.



(a) Task creation (b) Selecting a block

Figure 3.4: 3.4(a) shows the screen the end-user sees when he presses the button to create a task and then presses the left button with the plus symbol. 3.4(b) shows the screen that is shown after the end-user selects the sensor group in Figure 3.4(a).

called *quick action menu*[4] and it lets the user select one block type. After selecting one block category, a list showing the blocks is displayed (Figure 3.4(b)). Notice that there are two columns in Figure 3.4(a). Blocks are added into each column depending if the button touched was the left one or the right one.

A task is saved every time a user presses the back button or presses the save button on top.

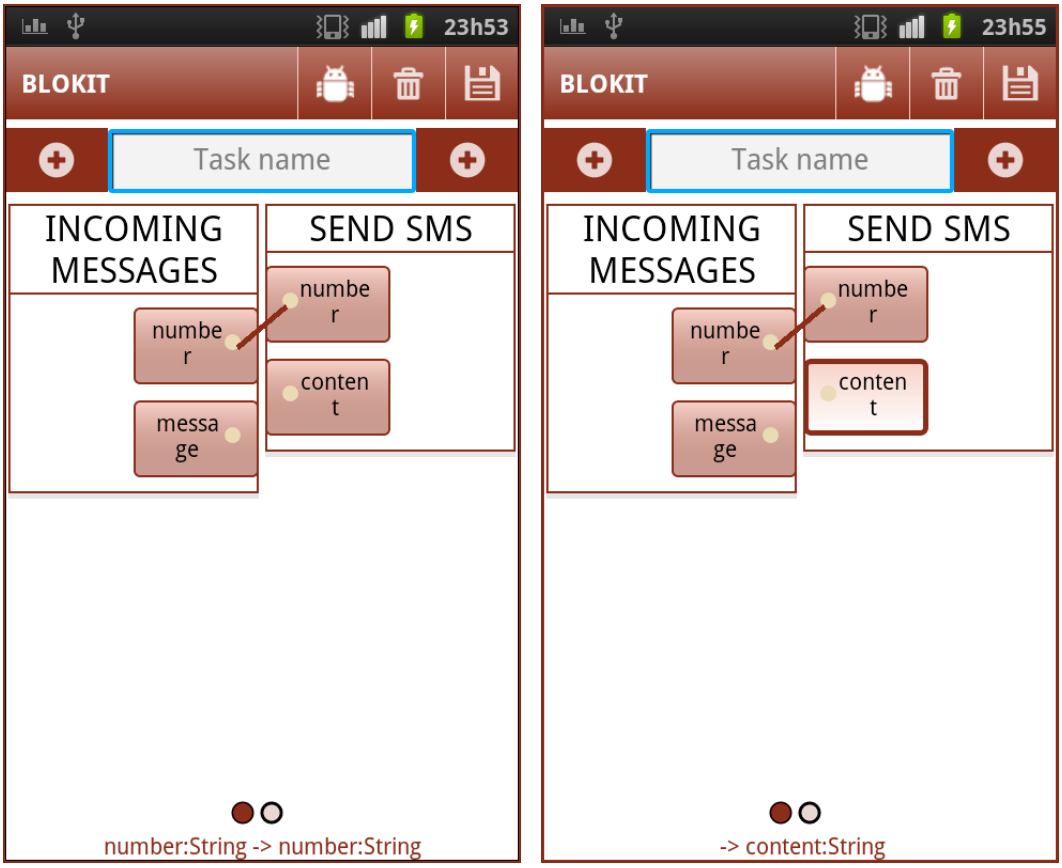
3.4.2 Connecting Connectors

A connection between an output and an input is displayed using a red line (Figure 3.5(a) (p. 34)). This line follows the scroll movement on both block columns. We added some features that help the end-user to anticipate some errors: small colored circles and a connection status message.

[4] Quick action menu is an Android pattern that is used for actions that are based in a context [dp].

Small Colored Circles

Each connector has a small colored circle that represents its type. This color is randomly[5] created every time the application starts. We choose to do a random color because we can't predict how many types the application will have. Therefore, we can't predict the amount of pre-defined colors that we need. However, each color is consistent in an application usage. Once they are generated for each type they keep the same color until the application is killed



(a) Connection example (b) A connection waiting to be finished

Figure 3.5: 3.5(a) shows a connection that was just made. Figure 3.5(b) has a connection waiting to be finished.

This colors create a visual type system. End-users can only connect outputs with inputs if they have the same color or if one of the connectors has a white color. White color is reserved to represent connectors that accept any type of input. One example of that input is a block that has a input called "value changed" [6]. This block can observe any change so it accepts all kind of values.

[5] Actually, this isn't completely random since we exclude the white color and we add some values in all RGB components so we try to get a bright color.

[6] This block is described in § 3.5.3 (p. 38)

Connection Status Message

The task creation activity has a message at the bottom of the screen that is updated every time an end-user selects a connector. Figure 3.5(b) (p. 34) shows a connection in standby, with following message: -> content:String. This means the end-user selected an input, at the right side, that this input has the type String. When a connection is successfully made we can see a message like: number: String -> number: String (Figure 3.5(a) (p. 34)). Outputs are updated in this message at the left of the symbol -> while inputs are updated at the right.

If a connection is not successfully a message will be shown to the end-user explaining the motive. This can happen when the end-user tries to connect two different types or the end-user tries to connect an output to an input that has a connection. An end-user can also make a wrong possible connection, and he can remove it by pressing the input and output of that connection.

It is possible to keep connecting more blocks to the right by *swiping*[7] from the right to the left. This way a new space will be created on the right and the list is shifted to the left. This gesture is only possible if a block exists on the right column.

3.4.3 Blocks' Operations

If an end-user presses a block, a quick action menu pops out with some options related to that block (Figure 3.6 (p. 36)). From this set of options it is possible to move the block to the right or the left. However, this will delete all the connections. The end-user can also see more info about the block or delete it. The info from a block has to be defined by the developer.

When a block was generated from a task a new option appears in the quick action menu, and it is possible to see that task. The end-user can go back to the task he was creating anytime pressing the back button.

3.5 BLOCKS IMPLEMENTED

We implemented a set of blocks for testing. We used our prototype a fair amount of times, and each time we used it we thought in ways to improve it. Unfortunately, creating new blocks was always at the bottom on our to-do list. Even so, we made 12 blocks that will be described within this section.

3.5.1 Sensors

As we described in § 24 (p. 27) sensors are blocks that provide data. Sensors usually depend on a set of *broadcast receivers* [Rec]. The type of broadcasts that are received by sensors usually are

[7] Swipe is a gesture that uses one finger. Swiping can be done from the left to the right or from right to the left. The gesture requires that the user touches the screen and moves the finger in an horizontal direction.

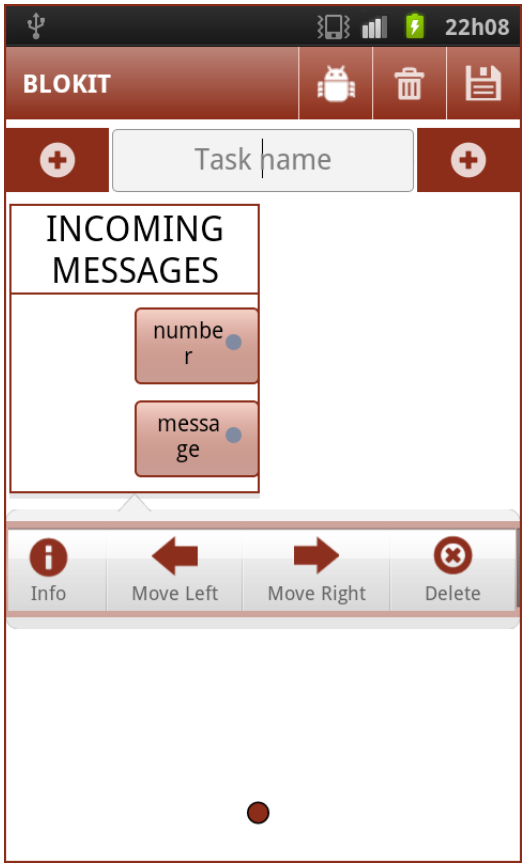


Figure 3.6: Operations that can be done on a block.

related with events that have some sort of change, such as: time changes, position changes or wifi status changes.

Time

The information in this block is updated every minute and provides info about the current hour and minutes of the system.

Battery Status

Battery Status sensor has one output that provides info about the percentage of the battery left. This information is managed by Android that sends a broadcast when a change in the battery level occurs. Battery status catches the broadcast and updates the respective output.

Incoming Messages

This sensor has two outputs that are updated when a message is received. One output is the number that sent the message, and the other output is the content of that message.

Network Status

This sensor is useful to know if a wireless connection is activated or not. Every time the wireless status changes the only output this sensor has it is updated.

Timer Sensor

This is an example of a configurable sensor. This sensor creates a timer in minutes that uses the value of its single input. When timer sensor receives new data, the Android operating system creates an alarm that is repeated for the amount of time received (and it cancels the other alarm created by this block). An alarm fires with a broadcast. When this sensor receives the broadcast it updates its output with the value "true", so other blocks know that an alarm was activated. Next this value is set to false.

3.5.2 Actuators

In § 24 (p. 27) we introduced the concept of actuators. An actuator can do anything, but usually it needs at least one input to fire its action.

Notification

A notification in Android has a title and a content and it can appear any time in the top bar. The notification actuator has two string inputs to fill those fields, and both inputs are mandatory.

Send SMS

The send SMS actuator has two required inputs. The destination number of the SMS and the SMS's content.

Vibrate

The vibrate actuator makes the smartphone vibrate. It receives an integer that says how much time the smartphone will vibrate in milliseconds.

3.5.3 Other Blocks

We created four main categories of blocks: *Sensors*, *Actuators*, *Input Providers* and *Others*. Despite "others" could be divided in other subcategories, we felt that we were over classifying for the amount of blocks we have. In fact, we also considered removing the input provider category. However, this category is important because it has a different interaction with the end-user. Therefore, all blocks that don't fit in the category sensor (they don't receive broadcasts);

or the actuators sensors (they don't execute actions); or the input providers (they don't receive input from the user) they belong to the "others" category.

Input Providers

We created two input providers for strings and for integers.

Changes Detector

Changes detector is a special block that has two inputs: An input that will *observe* an output (the changes controller/detector) and the input A. This block is inspired in the selector node that we saw in § 16 (p. 20) but it works as a filter and not as a selector. The main idea is: if this block receives a new value in the change detector input, then it fires A into its output.

Equal

The equal blocks works as a two-in-one block. It operates like a selector node that we saw in § 16 (p. 20) but the control value is an internal value. Basically, this is the same as having two blocks connected. One block that compares and fires a value that can be `true` or `false`, and this value is connected to a selector. If A is equaled to B, the selector forwards the true value to its output, otherwise, it forwards the false value.

Number to Text

This is a simple block that converts an integer to string. We called it "Integer to Text" after some end-users told us that they didn't know what a string was.

Sum

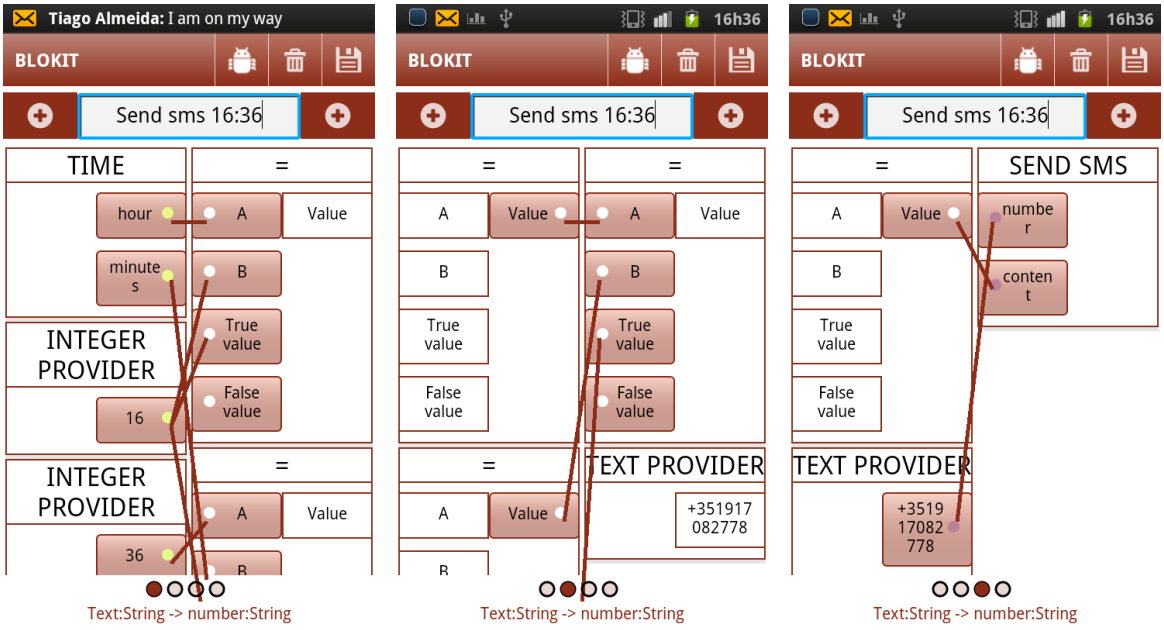
Sums two integer values and puts the result in the output.

3.6 A TASK EXAMPLE

All features that were described and all blocks that were made are useful to build tasks. In this section an example of a task will be given.

3.6.1 Building a Simple Task

Each task only shows two columns of blocks in the screen, therefore, to show a task, we had to take more than one screen shot. Each screen shot is taken after a swipe from the right to the left.



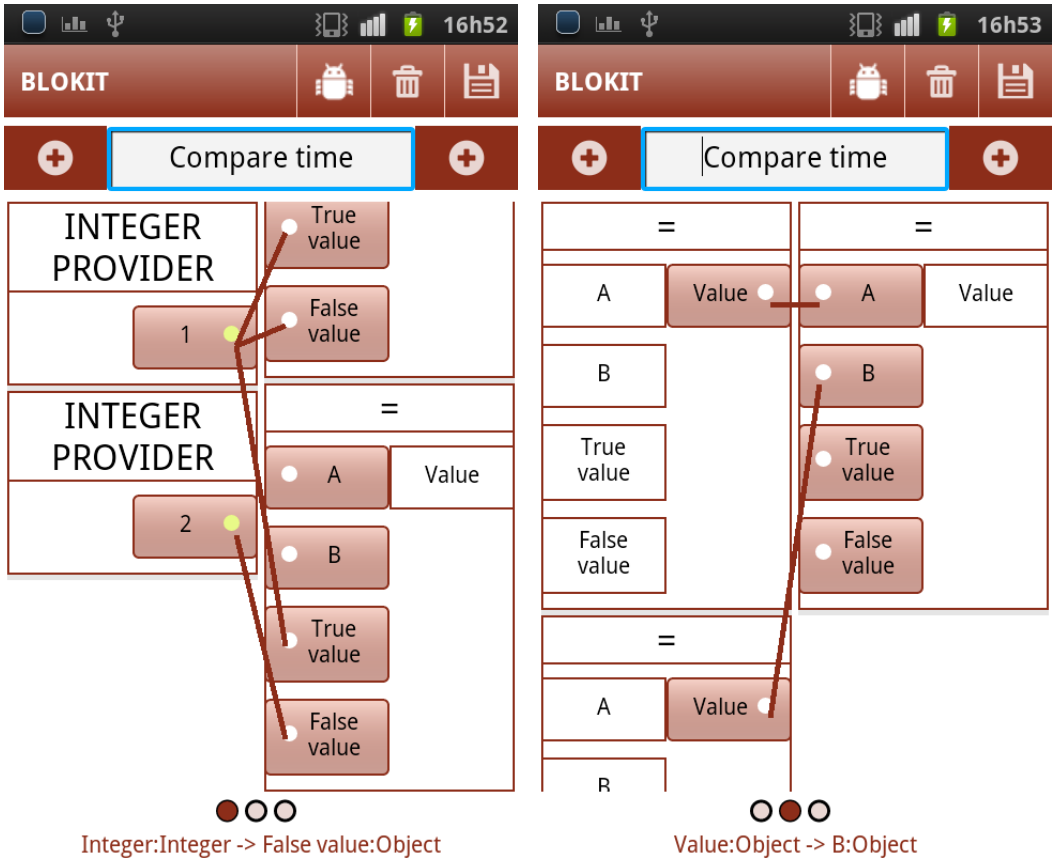
(a) Part 1 (b) Part 2 (c) Part 3

Figure 3.7: An example of a task that schedules a SMS to be sent at 16:36.

In Figure 3.7 we reproduce a task that an end-user tried to create. This task sends a SMS to the number of the smartphone where we are conducting this experience and with the content "I am on my way" at 16:36. Note the message being received, as scheduled, in the top of the Figure 3.7(a). That figure also shows that this task needs a true value in the first equals block and in the second equals block and then we need to compare the values from those outputs to check if both equal blocks are true. In this case we pass the value 16 for both equals block, so, if both are equal then both will have the 16 as their value on their inputs. This is needed because we don't have an "and" block, otherwise it would be simpler. Figure 3.7(c) also has the the content of the message connected to the true value input. However, the text provider block that connects to the true value is below the second equals block.

3.6.2 Simplifying With a Task Reuse

Those three equals blocks can be simplified if we create an auxiliary task. Figure 3.8 (p. 40) shows a task that can be used to simplify the three equals blocks used in Figure 3.7 into a single block. This block expects the hour and minute from the sensor and the values that should be compared, therefore the A and B values from the two equal blocks in Figure 3.8(a) (p. 40) have no connections. A true and false value can also be defined. This task can be reused to simplify the task defined in § 3.6.1 (p. 38).



(a) Part 1 (b) Part 2

Figure 3.8: An example of a task can be used to compare time.

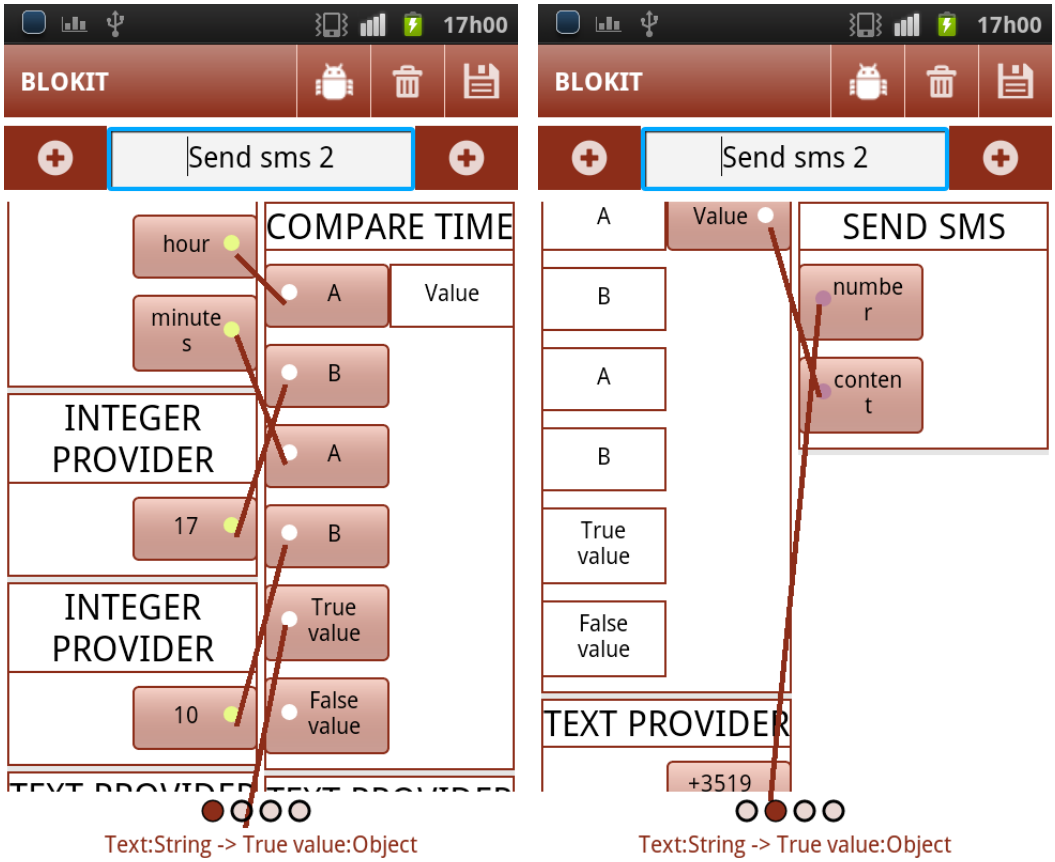
3.6.3 Building a Simpler Task

Figure 3.9 (p. 41) shows a simplification of the task defined in § 3.6.1 (p. 38) using the task defined in § 3.6.2 (p. 39). Although we named that task "Compare Time" the truth is that it could be called "compare two values" since it accepts any value in A or B. However, if an A (or B) is connected then the corresponding B (or A) has to be from the same type, otherwise an error is shown. Figure 3.9(b) (p. 41) also has the text "I am on my way" in a text input block, but that is hidden.

However, not all tasks are this straightforward to create, so end-users need debug methods to create more complex tasks.

3.7 DEBUGGING

Debugging is one of the problems that EUSE studies (§ 2.1.2 (p. 11)). We decided to add two forms of debugging: A common logging for all blocks and a visual form of debugging.



(a) Part 1 (b) Part 2

Figure 3.9: A task simplified that reuses the task defined in § 3.6.2 (p. 39) to build the task from § 3.6.1 (p. 38)

3.7.1 Logging

Each block can write any sentence in the global logging. Logging is accessible by the main menu, and each message has a time stamp that is set automatically, as seen in Figure 3.10(b) (p. 42). Messages at the top of the logging are more recent and the logging is updated in real time.

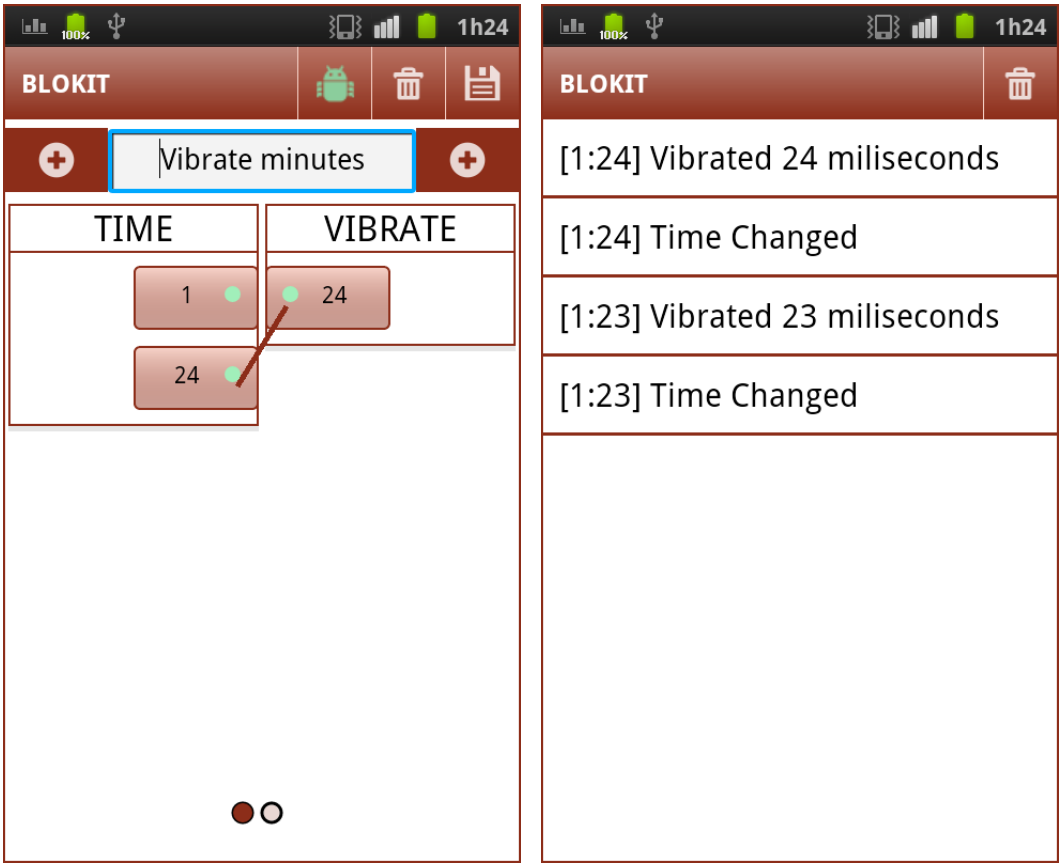
This debug method is useful for fast-changing actions or actions that can't be seen or felt. In Figure 3.10(b) (p. 42) there is an example of an action that is hard to feel: a vibration of 24 milliseconds[8].

3.7.2 Visual Debugging

When seeing a task is possible to activate a debug mode by touching the most left button at the top of the screen. In this mode the connectors' values can be seen instead of their name as we show in Figure 3.10(a) (p. 42). Those values are updated and propagated at real time.

This method is useful for end-users, so they can check if they are connecting the right values

[8] In fact, for developers who use an emulator, this action impossible to feel.



(a) Debugging in real time (b) Debugging with a log

Figure 3.10: 3.10(a) - debugging option in real time. 3.10(b) - debugging option with a log.

and if the blocks are providing the information they were expecting. We believe this solution will help surpass the understanding and information barrier (§ 2.1.1 (p. 9)).

3.8 AN EXTENSIBLE API FOR EXPERT PROGRAMMERS

We have an API to create blocks, and it can be used by expert programmers. The process of block creation is simple and it requires the definition of three methods: block constructor, new value and get description. Sensors are a special case that also needs to define the register method.

In the following sections, we will illustrate these methods using the time block that we saw in § 40 (p. 36).

3.8.1 Block Constructor

In the block constructor, the programmer should add the connectors of the block. Source 3.2 (p. 43) shows the block constructor of the time block.

Notice the call to the constructor with the name of block. We made this so programmers

don't forget they have to name their blocks. The Context [Con] in the constructor is needed in every Android block, because many of them require a context to do something (the vibrate block, for instance, needs a context).

```
1 public TimeSensor(Context cont) throws Exception{
2   super("Time", cont);
3   Output<Integer> hour = new Output<Integer>("hour", Integer.class);
4   Output<Integer> minutes = new Output<Integer>("minutes", Integer.class);
5
6   addOutput(hour);
7   addOutput(minutes);
8 }
```

Source 3.2: Block constructor.

3.8.2 New Value

The new value method is called every time an input's value changes or a sensor receives an event. In this case, the Source 3.3 is called every time the minute changes, and it updates the outputs' values with the current time. On line 2 we add a log that says "Time Changed". It is important to add the log before setting the values (lines 5 and 6), otherwise those values can propagate and executes other actions.

```
1 public void newValue(){
2   PublicData.addLog("Time Changed", this);
3
4   Calendar calendar = Calendar.getInstance();
5   outputs.get(0).setValue(calendar.get(Calendar.HOUR_OF_DAY));
6   outputs.get(1).setValue(calendar.get(Calendar.MINUTE));
7 }
```

Source 3.3: New value function that has to be defined.

3.8.3 Block Description and Sensor Registration

Block description and sensor registration work in the same way. Both need values that could be passed in the constructor, but we decided to make two abstract methods instead. This way, expert developers don't forget to define these values (otherwise they could generate the super constructor automatically with null values). Furthermore, on the first usage, since these methods are abstract, the compiler will force developers to define them.

The method getBlockDescription returns a String and the method register returns an ArrayList <String>. This ArrayList needs to have the strings that are the key for the broadcast the developer wants to receive. As shown in Source 3.4, we are registering for the broadcast "android.intent.action.TIME_TICK".

```
1 public ArrayList<String> register() {
2   ArrayList<String> res = new ArrayList<String>();
3
4   res.add("android.intent.action.TIME_TICK");
5   return res;
6 }
```

Source 3.4: Registration of the intents that this block wants to receive.

3.9 SOURCES OF INSPIRATION AND SOLUTIONS

We tried to provide solutions to the problems that were seen through § 2.1 (p. 9). Those solutions were inspired by VPLs (§ 2.3 (p. 16)) and by the data-flow execution model (§ 2.2 (p. 14). The combination of this two concepts usually leads to another one called VDFPL (§ 2.4 (p. 19)).

In this section, we will intersect those concepts with our prototype.

3.9.1 Data-Flow

We didn't want to create a pure data-flow language, but we tried to follow some of its properties. In Table 3.1 (p. 45) we will analyze each property of the data execution model (§ 2.2 (p. 14)).

We should also note that our graph isn't executed in parallel. Android has some limitations in terms of threads. In the beginning, we tried to do a thread for each block (so they can execute in parallel as soon as they get their data) but as soon as we reached more than three blocks (three threads) we felt some differences on the performance.

Most of the properties don't match with the data-flow execution model because: we are embedding our solution in an environment that doesn't follow the data-flow execution model, and we are using an imperative language (Java).

3.9.2 Visual Programming Languages

In § 2.3.2 (p. 18) we introduced a classification scheme for VPLs. We belive our solution is a *pure visual language* since the user can't see the difference between execution or compilation. We gave some steps in the direction of an *hybrid text and visual systems* with the task serialization, but we didn't finish this system. The main idea was: since all tasks can be serializable, we could save

Property	Analysis
Freedom from side effects	Outputs and inputs use independent values so, there are no side effects caused by variables. However, we can't control what each block does with the environment. Although a block can't change other block outputs directly, he can produce side effects like changing the alarm system, or even ring a tune.
Single assignment rule	This property fails in our scenario. Sensors and actuators are always changing their outputs, and they reuse their variables.
Locality of effect	In our case, we established a scope for every variable: a Task.
Data dependencies equivalent to scheduling	Sensors don't depend on other instructions but they depend on external events.
Lack of history sensitivity in procedures	Although the only historical data that is saved is kept in the connections, we can't prevent that developers save data on their blocks from one execution to another.

Table 3.1: Data-flow properties analysis

tasks in a XML format. Developers could create tasks with text in the PC and next upload it to the smartphone.

VPLs have a set of characteristics analyzed in § 2.3.1 (p. 17). In our case, the only characteristic that could be improved is the *concreteness*. We have a high level of abstraction because connectors can have any type, and block can be anything. We believe this level of abstraction can be an advantage in the long term because it can produce more solutions. However, we consider that this abstraction level requires that end-users do an additional effort to use our application.

3.9.3 Visual Data Flow Programming Languages

VDFPLs have a set of properties that were described in § 2.4 (p. 19). These properties are usually associated with a graph. In our scenario, nodes are blocks, and arcs are connections between nodes. We reviewed those properties looking at our application in Table 3.2 (p. 46).

We have no iteration constructs in our solution because e we don't allow outputs to be connected into inputs that were already processed. We only have one conditional construct called *equals*, and we already seen it in § 42 (p. 38).

3.10 INFORMAL TESTING

While we were developing our prototype, we tried to collect some feedback from end-users. That feedback changed some attributes in our application, and others were saved in a to-do list. We will group the end-users' feedback into two lists: a list with the items that were done and other that weren't done.

PROPERTY	ANALYSIS
"A visual data flow program code is presented as a directed graph."	A task is a directed graph since connections only have one direction: from outputs to inputs
"A node represents, for example, a simple computational unit or a subprogram consisting of other nodes. Also, a node can represent systems input or output value."	In our case, a node is a block. A block is a simple computational unit but also can be a task that was created using other blocks. We also have a block type that represents system input: Input Providers.
"A node can have zero or more input and output arcs."	In our case only sensors can have zero input arcs. The rest of the graph needs to have inputs, so they can be executed.
"An arc can be considered as a variable transferring data tokens from a node to another node or nodes."	An arc is a connection and data is always transferred from the output to the input.
"The arc is attached into the node through data ports or terminals."	We have our own name for data ports: inputs.
"The data type of an arc, and a token it transfers, has to be the same type as the input and output terminals it is attached to."	We have types in inputs, outputs and connections. However, we have a universal type (the Object, from java) that can be connected with any type.
"... a node becomes executable as soon as it has received all its input data and its (possible) output arcs are empty."	We execute a block as soon as it has all its inputs available. However, we don't guarantee that the block will only execute when the output arcs are empty.
"An arc transferring tokens into a node becomes empty immediately when a node has absorbed a token or tokens from it."	This only happens when we are using a connection with the type known as *last value*. If the connection has other types, the value is recorded. This value is only cleaned up when the block that receives the data wants.
"If a node does not have any inputs, it is immediately executed when the program is executed. Also, a node without any inputs is executed only once during the program execution."	There isn't a "execute" option in our prototype. A task is executed when something changes and it only has two type of blocks that generate values: Sensors and Input Providers. A sensor generates values based on events and Input Providers create values when an end-user changes its outputs.
"Once a node has been executed, it produces new values for all of its outputs."	We can't guarantee that. Developers can create a block that updates only one of its outputs.

Table 3.2: Visual data-flow programming languages properties analysis

3.10.1 Issues Tackled

Some issues noticed by the end-users were solved, and resulted in the following modifications that are part of our prototype:

- **Creating a Task**
  End-users, on the first usage, didn't know how to create a task. The fact is that the two-column layout isn't explicit enough, so we added a small tutorial (Chapter B (p. 81)). This

tutorial also tries to fill another lack: some end-users couldn't understand what a block was without an explanation.

- **Connector Types** Before our solution for types, we had no information about types in each connector. We thought that the connector's name was enough, but some names can be misleading. An example of a confusing name is the name number, that is used on SMS blocks. This connector is a String but some end-users thought it was an Integer. To solve this issue we added colors to the connectors and a message at the bottom of the screen § 3.4.2 (p. 33).

- **See a task Composition** To insert a task, it is necessary to access a category of blocks named "tasks". Some end-users tried to see what was the task composition, and they couldn't. So we added a button in the block's quick action menu. This button only appears if the block was created from a task.

- **Block Info** We thought that the block name, and the connectors' names were self-explanatory. However, many end-users asked us what each block did. We solved this lack of information with a small description in each block. This description is created by the developers.

3.10.2 Issues to be Solved

The following issues were noted by end-users but they are still open:

- **Blocks that need all inputs connected so they could work** End-users couldn't make some blocks work because they didn't know they needed to connect all inputs.

- **Edit directly an input** There were end-users which tried to edit a block input directly. We had to say they needed to use an input provider.

- **Collapse input providers** Some end-users suggested that a task with input providers could be simplified if an input provider could be collapsed into the input it is connected with.

- **Finding blocks** Some end-users couldn't find the blocks they were looking for and they tried to find them in wrong categories.

- **Conversors** Sometimes it is necessary to use a type converter. One of the common problems was the SMS number from the SMS receiver or the SMS sender blocks. We saw some attempts to connect an integer provider with the number on SMS sender block. We thought that end-users were expecting automatic conversions.

- **Complex schemes for simple tasks** End-users told us that the amount of blocks was too high for simple tasks. The example gave in § 3.6 (p. 38) (that an end-user tried) uses nine blocks to send a message at a scheduled time. Almost all suggestions that end-users have done could simplify the number of blocks needed. Removing some conversors, and simplify the input providers are examples of useful suggestions.

3.11 LESSONS LEARNED

A lot of time was invested thinking on how we could represent the blocks' info on the screen. If we create a script in Blender Node Compositor, that has 40-50 blocks, the connections start to overlap, some blocks get out of the screen and the script becomes messy. In a smartphone that could happen sooner. When we started to do some spikes in this subject, we found out that with three/four blocks in the screen the task was already messy. We needed a way to surpass that. We thought in a time-line that we could add blocks or even a canvas with draggable blocks that we could do zoom. Step by step, we created some concepts that were fundamental to the solution that was built.

A system with a clean interface ready to be used by end-users, was developed. We spent some time improving the interface, and fixing bugs, based on end-users' feedback. We soon realized that some concepts weren't as transparent as we thought. When we were reviewing the state of the art (Chapter 2 (p. 9)) we recognized that we were warned about this issue by Blackwell (§ 2.1.1 (p. 10)) [Bla06]. We tried to clarify some concepts with as tutorial (Chapter B (p. 81)). This tutorial can answer to most of the questions that were made by some end-users.

We believed we achieved a good result but, as soon as end-users started to try to build some tasks, we realized that we needed something more. A task that was suppose to be simple had more than five blocks.

We thought in a graphical rewrite rule system, where rules could be done by developers. We could use this system to simplify input providers. Whenever a user connects a output from a input provider with an input from other block, we could group those two blocks in a block that has a constant as input. This grouping could be done with a rule like $\text{Sum} + \text{Input Provider} \Rightarrow \text{Sum}$.

Another feature that would be helpful is type inference. In the equals block, for example, if an Integer was connected in A, then B input's type should be changed. Implicit conversion is also important. Implicit conversion could simplify some blocks. If the Integer to String conversion was implicit, we wouldn't need blocks to do this conversion.

3.12 CONCLUSIONS

Although block-based representation is not the best solution concerning such a small quantity of real-estate screen, it remains the most flexible way to represent the information we needed.

Our independent core from platforms allowed us to do automatic unitary tests. These tests were particularly important in the connection types (last value, FIFO or LIFO) since it was the only way to test them.

We can't say we have a VDFPL, but we can say that our solution was highly inspired by this set of languages. Although, we can't have some rules because our system can be developed by external developers. Another source of inspiration was the Blender Node Compositor.

No formal testing was made with end-users. In fact, almost all testing was made in an informal way. We had a device with the application installed, and we passed it to end-users. Then, we watched while they were trying to do a task that they imagined, and we tried to perceive the difficulties they found.

With this informal testing we learned the priorities of the end-users and we realized we could do some simplifications in the number of blocks used. Therefore, we decided to look on ways to do them before doing a formal testing with end-users.

Chapter 4

Experimenting other paradigms

Sometimes programmers don't realize that there are other programming paradigms that can be used. We wanted to build a rewrite rule system, but we didn't want to write more code than the necessary. When we were thinking in solutions for this problem we thought in a declarative language called Prolog and in a functional language that uses pattern matching called Scala. We may have increased the risk of failure, but in return, we increased the odds to learn something new, and we could show that there are other options beyond imperative languages.

4.1 MOTIVATION

The first round in our fight against Java started when we needed to use Java's generics. When Java was created it didn't have generics. Actually, the lack of generics led to some research like Pizza [OW97] and GJ [BOSW98]. Those researches ultimately led to the Java's generics that we have nowadays, and they were part of the JES2 5.0 that was launched in 2004. Scala's creator was involved in those researches, so it gave him more bases to develop the Scala language [Oa04].

Java's generics were needed for the connectors' types, and after some usage of this Java's feature we stumbled in the main problem with Java's generics: *type erasure* [Era]. Type erasure is a technique that converts any class to its raw type. The class ArrayList<Type>, for instance, is translated into ArrayList. With this technique, all information about the types that a class is using gets lost in the compile time. In our case outputs and inputs are created using Output<Type> or

Input<Type>. It was necessary know what type inputs and outputs are using so we can check if they can be connected or not. This necessity led to some experiences in Java so we could try to determine connectors' types at runtime.

To solve the problem caused by type erasure, we tried to use *Java's reflection* [Ref]. Java's reflection provides methods to get information from classes and objects at runtime. We tried some solutions to discover connectors' types in runtime using Java's reflection. Despite most of them work of them start to fail if a developer creates a new class that extends Output<Type>.

We ended up with passing the connectors' type as an argument of the constructor like this: Input (..., Class<?> classType, ...). This solution was being avoided since we started the experiments, because it introduces overhead in the API. With this solution we didn't feel that we won this battle.

We beat a challenge (even with a non-optimal solution) but there were more challenges waiting for us. When we started to test the application, we realized that users tried to make connections that made some sense, but since we had a strongly typed solution, they couldn't do it. An user could try to connect the minutes from a time sensor to the content of a message, and it wouldn't work because minute is an Integer, and the message's content expects a String. However, in Java, it will throw an exception if we try to set a value in a String using an Integer, so we only allow connections from the same type[1].

Automatic conversion could be solved for some cases, but only solving for specific cases isn't enough. Furthermore, these simplifications weren't enough, we needed something more powerful that could simplify automatically a set of blocks into a single block: a *rewrite rule system*. Imagine that an end-user has an integer provider connected with a sum. Why not simplify those two blocks into a block that had a constant input that is going to be summed to another input?

This rewrite rule system combined with the curiosity of experimenting new paradigms made us look for other programming languages. We only looked for programming languages that could be connected with the work we done because we didn't want to delete all our work and start again. We wanted to improve it.

4.2 A VISUAL REWRITE RULE SYSTEM

When we started to see some rules with more than five blocks, we saw that some simplifications could be made. However, we didn't believe that end-users would create small tasks, so they could simplify some tasks. When the tasks reuse was conceived, we thought that only tasks with some complexity (or usefulness) would be reused.

An automatic mechanism to do some simple and smart simplifications was needed. Random blocks couldn't be grouped, otherwise it would exist cases with a cluster of big blocks without any

[1] Although it is possible to use blocks that convert types, sometimes it is useful to have automatic conversions to remove this visual overhead

kind of meaning (grouping all sensors into one sensor, for instance, with a lot of outputs that had nothing to do with each other). So, we thought in a visual rewrite rule system with rules that could be written by expert programmers. These rules have a left term that we call the *matching term*, and a right term that we call *resulting block*. The right term, for simplification purposes, has only one resulting block, and it has a block name. If a matching term is found a resulting block is created.

4.2.1 A Set of rule types

We thought in some useful rules and how they could be represented:

- Block1 + Block2 ⇒ Block3
  If a block with the name "Block1" is connected to a block with the name "Block2", then create a block named "Block3" that groups the two blocks. Example: IsEqual + Selector ⇒ Equals

  IsEqual is a block that checks if two values are equal and provides a boolean as output. Selector is one block that was seen in § 2.4 (p. 19). It has a control value that chooses one input from the true or false input.

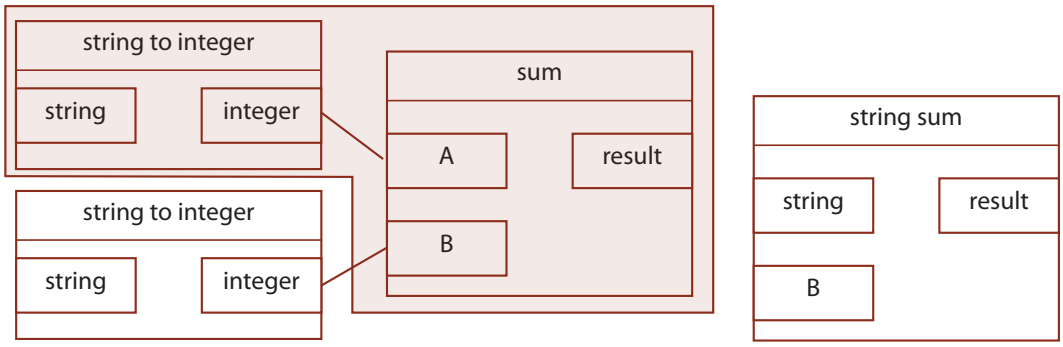
- Integer*Block1 + Block2 ⇒ Block3
  If a number of blocks with the name "Block1" is connected with "Block2", then group those blocks into a block called "Block3".

- ?*Block1 + Block2 ⇒ Block3
  Groups all blocks with the name "Block1" with "Block2" and produce the "Block3" that has attached the number of blocks with the "Block1" name who were grouped.

- Integer*Block1 + ? ⇒ Block1
  If a number of blocks with the name "Block1" is connected with any block, then create a new block that groups all blocks with the name "Block1".

The character "?" can be used in the matching term in two ways: With a block through the use of the "*" symbol or the "+" symbol. If it is being used with the "*" symbol, it means that we want all blocks that are connected with the next block. If the character "?" substitutes a block it means that it can be any block and is used with the "+" symbol.

The matching term is used to collect a set of blocks that fit that rule. After this set of blocks is gathered they are grouped into the block from the right term.



(a) Group selected (b) Grouping result

Figure 4.1: The figure shows a selection and the grouping for the rule string to integer + sum ⇒ string sum. 4.1(a) shows what was selected and 4.1(b) shows the result of that selection .

4.2.2 Grouping Blocks

Imagine that we want to group the blocks using this rule: string to integer + sum ⇒ string sum. The first thing we are going to do is gather a set of blocks that match the matching term. Figure 4.1(a) shows an example of a block composition and the blocks that were selected.

After we gather all blocks we have to select which connectors will be used in the grouped block. A connector is selected if: has no connections **or** has a connection from a block that is outside the matched group. Figure 4.1(b) shows the result from the example in Figure 4.1(a). Notice that the input B didn't change in the grouped block. This happened because this input B is connected with a block that isn't part of the blocks that were grouped.

To build this rewrite rule system we tried to use Prolog with an engine for Java. Prolog is a declarative language that is commonly used in artificial intelligence [Wik12]. We had already used Prolog in the past so it didn't require a lot of effort to learn this language. Additionally, Prolog provides a kind of pattern matching that is useful to build this set of rules.

4.3 A DECLARATIVE LANGUAGE: PROLOG

The idea of Prolog came when one of us said that there was an engine that could run Prolog in Java. We became interested and we made some small tests[2] and we reacted with enthusiasm to some "true" sentences.

Some coding was done with Prolog using SWI-Prolog. After some experiments that were made to review some concepts, we made a simple rewrite rule engine, and we tried to connect it with Java. We got some problems with three engines, and we decided to forget Prolog.

[2] We used JUnit for those tests. We basically wanted to check if we could run simple facts on the Prolog engine

4.3.1 Some Experiments

Prolog allows the user to describe relations using two structures: *facts* and *rules*. Rules are used to make conditional statements [Rul] and facts are rules that always return true.

Two facts were created in Prolog to map the information that Java had[3], and then we could ask for a list with the new configuration depending on the rules that were defined.

The two types of simple facts that were created are: block and connection. These facts only had the basic information and are represented in Source 4.1. Since each block has an Id, in Java, we could easily connect the Prolog's facts with Java.

```
1 block(intprovider, 1).
2 block(intprovider, 2).
3 block(sum, 3).
4
5 connection(1, 3).
6 connection(2, 3).
```

Source 4.1: An example of two integer providers connected to a sum using Prolog.

Rules were declared using a predicate. A rule returns the blocks that match with the matching term, and the outputs of the resulting block. Then, we just need to apply a `findall` to see which rules would apply and save all group blocks on a structure. At last, we needed to update the facts database dynamically with the new group blocks and delete the blocks that we used.

This solution was incomplete, mainly because: we had no general way to declare rules (an expression system, for instance), and we were not saving the inputs and outputs of the group block.

Before this solution could be completed we decided to integrate with the prototype, and some problems were found.

4.3.2 Problems

After some debugging and fixing some errors, we had something that was working in Prolog. However, we wanted to integrate our work with Java. When we tried to integrate, the problems started.

The first engine that was tested was the W-Prolog engine [WP], after this, we tried tuProlog [DOR05] and at last, when we were about to give up we tested Jekejeke Prolog [Jek]. The experience we had with these engines was listed:

- **W-Prolog**
  When an error occurred (parsing, per example) usually there was no feedback. An exam-

[3] Although, after this was done we realized we could create a List with the data and pass it into a Prolog predicate

ple of one error we had that took some time to be fixed was: W-Prolog couldn't run our Prolog code because there was a fact with an underscore in a variable (something like this: `block(input\_provider, 1)`). We removed the underscores, but more problems appeared. W-Prolog doesn't accept the command `:- dynamic binding`, and it is needed so we could add and remove facts dynamically.

- **tuProlog**
  We couldn't work with this engine because the error feedback was also poor. Furthermore, when we had a predicate that was returning several options[4] and we couldn't manage to get a single answer from the engine because at the end of those options the result was "false"-

- **Jekejeke Prolog**
  The errors' feedback in this engine were good. We had to make some adjustments because this engine didn't recognize some functions that weren't Prolog native. After some changes, our Prolog code was working but we realized that there was a lot of work missing. We needed to parse the results from the resulting `List`, and that `List` was returning more results that we intended. This engine made us realize that we had a fair amount of errors in the Prolog side.

When we realized we had to sync our data with Prolog, every time the user would do an operation on the interface we asked ourselves: Prolog is the only option we have? Does this simplification using Prolog is worth it?

4.3.3 Leassons Learned

Although finding an engine that could fit our work was hard, we finally had our experiments working with Jekejeke Prolog. However, we spent more time than expected in remembering Prolog and testing and discovering engines. At the time, we had something working, but we realized that there was more work to be done, and it was going to take some time to do it. Furthermore, we re-analyzed the Prolog option when we noticed the amount of overheads we were adding.

There were one more option that that we were ignoring: Scala. This option was been ignored because we believed that Scala has a steep learning curve. After we considered this option again, we decided to look after some ways to learn Scala. We ended up reading parts of a book of Martin Odersky, the creator of Scala [OSV08]. This book had all information that we needed to learn this language, so we felt we had to give a chance to Scala.

[4] In Prolog, sometimes, more than one solution is returned. When that happens it is possible to use the character ";" in the Prolog console to return more solutions.

4.4 AN OBJECT-FUNCTIONAL LANGUAGE: SCALA

> *"The name Scala stands for "scalable language"... It runs on the standard Java platform and interoperates seamlessly with all Java libraries... Technically, Scala is a blend of object-oriented and functional programming concepts in a statically typed language." [OSV08]*

Scala it is a recent language that was released late 2003 / early 2004 [Oa04], and since then many known names already used Scala (like twitter [oS]) and others are just starting (like Wolfram Alpha [dap]).

Scala programs can be run in the Java Virtual Machine (JVM) allowing developers to add value to programs that were built using Java.

It was hard to start working with Scala, but as soon as we started to go deeper in the Scala language, we found some advantages over Java.

4.4.1 Why Scala?

Instead of providing a "perfectly complete" language, Scala creator provided means so the Scala users could easily expand the language with a built-in syntax. This motivated developers to create libraries.

Scala has two important features that can help us improve our work: *pattern matching* and *implicits*. Pattern matching works like `switch` in Java but it is more powerful than a `switch` because we can compare all sort of objects. Implicits are useful to convert one type to another depending on the context. Pattern matching can be used for the rewrite rule system and implicits can be used to provide automatic conversions between types.

APIs that can Look Like Built-in Syntax

All operations that come with the Scala language are methods. Operations like `for`, `while` and `if` are methods. The syntax of these operations are similar to Java but they are, in fact, methods.

When we do `a +b`, what is actually being done is a call to a method with the name "+"[5]. Since we can define operation with special characters and all methods with one parameter can be called without points, we can do code like `input >> output` to connect an input to an output, or `Block+Block` to group two blocks.

Adding Value to Existing Code

> *"Scala doesn't require you to leap backwards off the Java platform to step forward from the Java language. It allows you to add value to existing code... Scala code can call Java methods, access Java fields, inherit from Java classes, and implement Java interfaces... Scala code can also be invoked from Java code." [OSV08]*

[5] The same as: `a.+(b)`

Scala code uses Java types and improves its syntax through the use of implicits. An example is the conversion between String and Integer. Instead of using Integer.parseInt(string) we can do string.toInt. This is only possible because there is an implicit conversion that converts a Java's String to an instance of the Scala class StringOps[6]. This implicit conversion occurs when the Scala compiler tries to find the method toInt in the class String and it fails. After failing the compiler tries to see if there is an implicit that converts from String to a type that has the toInt method.

Implicits

> *"Scala's ... implicit conversions and parameters ... can make existing libraries much more pleasant to deal with by letting you leave out tedious, obvious details that obscure the interesting parts of your code." [OSV08]*

Implicits are useful to convert one type to another they can be used to convert ints to doubles or even ints to strings.

An implicit is declared using the keyword implicit.

```
1 val int: Int = 5
2 val string:String = int
```

Source 4.2: This code will fail at the second line giving an error.

If we try to execute the code showed in Source 4.2 it will prompt an error message on the second line saying:

```
1 error: type mismatch;
2 found   : Int
3 required: String
4       val string:String = int
```

If we define the implicit showed in Source 4.3, the code in Source 4.2 will run without problems. The compiler tries to execute the code then, before it gives up it checks for any implicit that could convert from Int to String.

```
1 implicit def intToString(i: Int): String = i.toString
```

Source 4.3: An example of an implicit that converts from Int to String.

In our case, implicits are useful because we can avoid using some blocks that convert data, since a developer can create implicits in a connector.

[6] StringOps has a method toInt

Pattern Matching and Case Classes

Pattern matching works like switch in Java but it is more powerful than a switch because it allows comparisons with any object type.

A class can be used with pattern matching if it is a *case class*[7] A case class is a class that has some work done by the compiler. The compiler creates:

- a factory method with the name of the class so classes can be created without using the new keyword
- all parameters declared within the list after the name of the class are automatically converted as fields of the class
- a constructor that expects all fields declared within that list
- the methods: hashCode, equals, toString and copy

Case classes are useful for pattern matching and also can simplify some coding.

An example of usage of pattern matching is arithmetic rules. Imagine that we had a set of case classes defined in Source 4.4. In that example, we have one expression at line seven that represents 2+3+0. That expression could be simplified automatically using pattern matching to 2+3.

```
1 //Case classes that can be used to build expressions
2 abstract class Expr
3 case class Number(num: Double) extends Expr
4 case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
5
6 //Declaring the expression 2+3+0 using the case classes
7 val expression = BinOp("+", Number(2), BinOp("+", Number(3), Number(0)))
```

Source 4.4: Definition and usage of the case classes Number and BinOp.

We can create a method called simplify that expects an expression and returns a simplified expression. That function is defined in Source 4.5 (p. 60) and simplifies the cases where the number zero is summed or the number one is multiplied. The match clause can be seen as a switch in Java but instead of switch(expr) we have expr match. If a match happens with a BinOp that has the operator "+" and the right side is a number with the "num" equal to 0 then we simplify the left side.

The last case works as the default in a switch, and the underscore means that all patterns can be matched. This default case simply returns the expression.

[7] There is another way to use classes in pattern matching through the use of extractor objects [oSEO] but it isn't as simple as case classes.

```
def simplify(expr: Expr): Expr =
  expr match {
    // Adding zero
    case BinOp("+", e, Number(0)) ⇒ simplify(e)
    // Multiplying by one
    case BinOp("*", e, Number(1)) ⇒ simplify(e)
    case _ ⇒ expr
  }
```

Source 4.5: Definition of a method simplify that simplifies expressions that have +0 or *1.

In pattern matching breaks aren't needed because all cases are analyzed in the order they are defined and each expression after the ⇒ symbol returns something. A match always expects returning values.

Case classes contribute to another property of Scala that we believe it is important: concise high-level code.

Concise High-level Code

> *"Scala programs tend to be short. Scala programmers have reported reductions in number of lines of up to a factor of ten compared to Java. These might be extreme cases. A more conservative estimate would be that a typical Scala program should have about half the number of lines of the same program written in Java." [OSV08]*

Case classes already supply an easy and concise way to create classes, but there are other properties that contribute to reduce lines of code.

Since Scala is high-level and functional we can apply functions to each list element in a concise way, and do things like: `name.exists(_.isUpper)` to check if there is any char in a upper state. Also, we can create functions that accept functions as an argument and simplify some code.

Another feature that simplifies the Scala code is the absence of the explicit declaration of types. Most of the times Scala can infers the types automatically.

Libraries are also a good way to save some code. Libraries exist in every language, but since Scala was conceived to be scalable it has some tools (like multiple inheritance[8]) that can enrich libraries so they can be more flexible.

Functional Philosophy

Scala's functional part gives tools to simplify some code, but it also shares some properties with the data-flow execution model.

[8] Multiple inheritance, in Scala, doesn't have the diamond problem (or the "deadly diamond of death") in multiple inheritance [Mar97].

Whiting affirmed in 1994 that "functional languages are a superset of data-flow languages" [WP94]. Although Scala isn't a pure functional language, Martin Odersky invites Scala's users to always try to use the functional paradigm.

We thought that if we tried to use a functional paradigm we would be more close of a data-flow language.

Libraries

Scala is recent, but since it was built to be scalable, Scala programmers already developed some libraries that integrate seamlessly with the language.

Akka [Akk] has some useful tools for data-flow, and it is one example of one library that can be used. Furthermore, there are good testing tools like ScalaCheck [Use] that allow us to run a set of tests. ScalaCheck also provides another way to do tests, where we declare a property and the test tries to find a counter example. We used this library to test our core and our rewrite rule system.

4.4.2 A Core in Scala

We first tried to use our Java's core in Scala as a library. We soon realized that without case classes, it would be harder to apply pattern matching. In addition, we felt that we had to start with simple tasks, and that we could simplify our code using Scala's Lists, so we re-created our core using Scala.

Our core architecture is almost the same as shown in § 3.3 (p. 29) at Figure 3.2 (p. 30). We have a class named `Connector` that is abstract. `Input` and `Output` are case classes that extend from `Connector`. `Actuator` and `Sensor` classes are also abstract.

However, we had to make some modifications to use case classes. Those modifications actually made sense, and we realized we could improve the Java's core.

Since all constructors expect all fields[9], we can automate some info that wasn't possible with our core in Java. In Scala we don't need methods like `addInput` or `addOutput`. Those changes together with the data transformation to Scala's Lists simplified some of the core's code.

Another interesting thing about the usage of case classes is that we can build tasks in a transparent way. An example of a task created for testing is given in Source 4.6 (p. 62), were task[10] that sums two numbers is created. The first argument of the `GroupBlock` constructor is its name ("GroupedSum"). The second argument is a List of Blocks. There are two ways to create a List in Scala: we can join each element with the `::` operator and finish with a `:: Nil`[11] or we can create it using the `List` constructor (`List(Block1, Block2, ...)`.

[9] A block constructor, for instance, expects the inputs and outputs that the block has

[10] A task is a group of blocks.

[11] Since the :: operator is related with Lists we need to apply it to a List, and therefore, we have to use it with an empty List (Nil).

Our List of blocks has two IntProviders that have two outputs (connectionA and connectionB) that were previously created. Also this list has a Sum block that has two inputs (inputA and inputB) and an output (outputB). The two last arguments are the inputs and outputs of this GroupBlock. At the time we made this core, we didn't have the rewrite rule system. If we had it, the last argument was not necessary since inputs and outputs lists could be calculated using the List of blocks.

```
1 val sumAB =
2   GroupBlock[IntEvent]("GroupedSum",
3   IntProvider(connectionA) ::
4   IntProvider(connectionB) ::
5   Sum(inputA, inputB, outputB) :: Nil, null, List(outputB))
```

Source 4.6: Creation of a task in Scala that sums two numbers.

4.4.3 A Rewrite Rule System

The rewrite rule system was one of the main reasons why we considered Scala. We had to define some new methods for this system, and we created a simple expression system (to represent the rules). We wanted to test the concept with two or three rules, and then expand it using our expressions system.

Expressions System

We conceived a way to express rules internally so they could be loaded from files[12]. Source 4.7 represents the possible combinations of the syntax in the expression rule system[13].

```
1 Produces → Expr ⇒ BlockName
2 Expr → BlockName | Join | Produces | Multiplier
3 Join → Expr + Expr
4 Multiplier → Number ∗ BlockName
```

Source 4.7: Definition of the syntax of the expression rule system.

It is easier to see how Source 4.7 can be used with an example. If we want to generate the expression 2*IntProvider + Sum ⇒ ConstantSum we could apply the steps in Source 4.8 (p. 63).

To prevent invalid syntax we start always with the Produces token. The numbers on the right tell us the line number of the rule (from Source 4.7) that was applied on the previous step. In the example, the Produces' token was selected and since the only option is Expr ⇒ BlockName we

[12] The part where we load from files and parse the information was not done
[13] We used Chomsky Normal Form [The] as a source of inspiration to this representation

```
1 Produces
2 Expr ⇒ BlockName (1)
3 Join ⇒ ConstantSum (2)
4 Expr + Expr ⇒ ConstantSum (3)
5 Multiplier + BlockName ⇒ ConstantSum (2)
6 Number*BlockName + Sum ⇒ ConstantSum (4)
7 2*IntProvider + Sum ⇒ ConstantSum
```

Source 4.8: Steps necessary to generate the rule 2*IntProvider + Sum ⇒ ConstantSum.

replaced the Produces token with that option. Then, BlockName can be replaced by a string and the Expr can be replaced by the Join token using line number two. These steps are repeated until we find a condition that only has BlockNames or Numbers. Then, we can replace BlockName by a String and Number by an Integer.

Showing like this is easy to see that our system prevents invalid rules in the syntax. We can't have things like BlockName or even BlockName + BlockName. This system was easy and fast to define, and resulted in the classes showed in Source 4.9.

```
1 sealed trait Expr
2 case class BlockName(name: String) extends Expr
3 case class Join(left: Expr, right: Expr) extends Expr
4 case class Produces(left: Expr, right: BlockName) extends Expr
5 case class Multiplier(number: Int, block: BlockName) extends Expr
```

Source 4.9: The classes created using Scala to produce expressions.

As defined in the Scala's documentation, a sealed trait[14] "may not be directly inherited, except if the inheriting template is defined in the same source file as the inherited class. However, subclasses of a sealed class can inherited anywhere" [oSSC].

A simple rule like 2*Integer Provider + Sum ⇒ SuperSum can be written in Scala like is shown in Source 4.10.

```
1 Produces(Join(Multiplier(2, Class("Integer Provider")), Class("Sum")),
    Class("SuperSum"))
```

Source 4.10: The classes created using Scala to produce expressions.

[14] Although the documentation points to a seal class, a seal trait is the same. A trait in Scala is similar to an interface in Java but in a trait it is possible to define default implementations for some methods.

Pattern Matching in the Engine

The rewrite rule system works with the following rules:

1. X*Block1 + Block2 ⇒ Block3
2. Block1 + Block2 + ... ⇒ Block3

Notice that the second rule accepts any number of blocks connected. First we made that rule with only two blocks and then we expanded it.

Some methods were needed to create the rewrite rule system. The most important one is the createGroupBlock. This method discovers the connectors of the group block, and uses them to create a group block.

The rewrite rule system has two parts. One part reads the matching term of the rule and tries to find situations where the rule matches, and the other that concludes the process, creating the block from the right side. Each part is a case in pattern matching, and the first part has another pattern matching inside. The second part checks if the expression was all analyzed and calls the createGroupBlock method.

To test the rewrite rule system and the core we used ScalaCheck [Use].

4.4.4 Testing all with ScalaCheck

ScalaCheck can be used to create unitary tests.

A set of tests starts with a name followed by the word should. Each test has a name that usually is used to represent what the test is going to verify. The name is followed by ! check, and then the test is declared.

ScalaCheck also provides another way to do tests through the use of *properties*[Use]. Properties can be declared and ScalaCheck will try to find a counter example where that property fails. An example of this kind of test is shown in Source 4.11. In this test, we are testing the associative property of an integer. ScalaCheck is going to try to find an i, j and z that is a counter-example to the property defined ((i+j) +k ≡ i +(j +k)).

```
1 "satisfy associativity" ! check {
2     forAll { (i: Int, j: Int, k: Int) ⇒ (i + j) + k ≡ i + (j + k) }
3 }
```

Source 4.11: A test that is going to try to find a counter-example.

Standard tests and tests that use properties were made in the core and in the rewrite rule system where blocks are created, connected, and a rule is applied. Then, the resulting group block is compared with a group block that we were expecting.

4.5 CONCLUSIONS

Scala is a language that takes some time to get used to, and every single thing we do with this language, it always has a way to be improved. The time we invested learning Scala always followed by more time improving our solution.

We decided to integrate what we have done, and we couldn't use it. In fact, we invested some time trying to integrate our work with Scala. We tried to call our core functions in Java, and it wasn't working. We tried to compile Scala to a jar, and we even tried to use bytecodes directly, but any of our attempts didn't work, and that was odd since the compiler wasn't throwing any errors. The errors happened in run time. Actually, we even ran an experimental code before we started to develop to Scala. However, that code didn't involved Android. That was our mistake.

The Android virtual machine is different from the JVM. Although Android SDK uses the Java language, Android uses a different virtual machine called Dalvik virtual machine (DVM). Android SDK compiles the source and resources, and converts them to run in the DVM. In theory, what we were attempting should have worked. However, the DVM was always throwing error saying it couldn't find our Scala classes.

We didn't find an answer for our problem, but we found other ways to work with Android. There is a plug-in for Scala that allow us to compile for Android. This plug-in is called android-plugin and it can be used to generate an apk[15]. The apk size is bigger than it is needed, the file is send directly to the device, there are no debug options besides a log printed in a console and it takes more time to compile than expected. Also, we couldn't use Java code with this plug-in, so if we want to use it was necessary to redo all the GUI without good debugging tools[16]. We didn't have time to take more risks, so we decided to exclude our work in Scala out of our prototype.

Scala is a recent language and it still lacks of mature development tools. Although there are already good tools like *Scala Building Tool*[17] [SBT] (SBT), we felt that Android development is hard with Scala, and the tools need to be improved.

Scala was a good experience and we believe that our work will be useful. Our work gave some ideas of what Scala can do, and it can be used in other platforms that use the standard JVM.

[15] Apk is the package used in Android for applications

[16] In this case we are referring to debugging with breakpoints. This is a tool that was often used in the development of our prototype.

[17] SBT is a tool that we used to compile/test and manage the dependencies of Scala

Chapter 5

Conclusions

This thesis touched several areas that deal with high levels of abstraction, and at the same time it had to be concrete, because it deals with end users. The end-users that used our prototype asked for higher levels of abstraction, so we made a detour that led to some experiences with Scala. Despite we didn't use the work we did in that detour, we believe that this was important. In this chapter, we are going to summarize our experience, our contributions, and our thoughts for the future work.

5.1 OVERVIEW

In Chapter 1 (p. 1) we introduced our thesis, and we presented some important concepts that are related with our thesis, such as: EUP, EUSE and VPL We also explained why we think this thesis is important, and the reason we choose to do a prototype for Android (first we explained the smartphones' choice then we explained the Android's choice).

Chapter 2 (p. 9) gives an overview of the problems that are usually faced in EUP and EUSE. Some of those problems can be solved using a VPL so we tried to understand the properties of this set of languages. Data-flow languages were also analyzed since they have a relation with VPLs. In fact, when this relation is strong, it can result in a VDFPL. In this chapter, we also saw two examples of applications that do something we were looking for: Tasker and Blender Node Composition.

The Chapter 3 (p. 25) is responsible for the information about our prototype. We explain how we organized our solution, what we have done and we finish that chapter with an analysis of our solution considering the concepts we saw in Chapter 2 (p. 9).

At last, in Chapter 4 (p. 51) we documented our experiments with other programming paradigms. These experiences were motivated by the need to simplify tasks, since it was one of the main faults identified in our prototype by end-users. We made experiences using Prolog, and then we made a dive in Scala. At the end, we couldn't integrate our work with the prototype, since we didn't find a mature way to work with Android and Scala.

5.2 MAIN CONTRIBUTIONS

We believe we have three major contributions: A short-paper, a framework that is independent from Android and a prototype developed for Android that can be used to collect more feedback from end-users.

5.2.1 Framework

This framework has a core that is independent from a platform, as long as the platform runs Java. The core allows the creation and connection of blocks and it automatically manages the data propagation between blocks. The connections can hold more than one value and those values can be requested by blocks.

We have the framework for both Java and Scala, and in Scala, we also have a small rewrite rule system.

The framework doesn't balance the abstraction level but provides tools so this level can be balanced. We think the block concept is enough to achieve high levels of abstraction that are good for end-users. However, it can also have low levels of abstraction. This level is managed by both expert programmers and end-users. End-users can increase the abstraction level since they can define tasks that can be used as a block. However, trusting the abstraction level to its end-users can lead to tasks that have more blocks than expected, so we decided to build a rewrite rule system that could do automatic simplifications.

5.2.2 Prototype

We believe that a typical visual data-flow solution with zooming isn't enough. We made a spike with draggable blocks, and with three-four blocks in the screen the blocks schema was confusing. So we decided to show only two blocks horizontally in the screen, leading to a solution with two columns that are scrollable. We also thought in limiting the number of vertical blocks since it can be confusing when five-six blocks on the left column connect to five-six blocks on the right

column. However, with the tests we made with end-users, no one ever used more than three blocks vertically.

Two forms of reuse were created: task sharing and task reusing. Task sharing allows an end-user to share a task he created so **others** end-users can reuse it. The second reuse form is achieved through the reuse of the tasks as blocks. Every time a task is created a new block type is automatically added to the set of usable blocks, so it can be used by the end-user.

Our prototype is far from finished, but it is useful to do more tests. It can be used to check what kind of blocks the end-users need, if there are other ways to connect blocks, and how easy is for end-users to create abstract tasks. These are only examples of tests that can be done using our prototype.

5.2.3 Short-paper

Our short-paper was submitted for the 9th international conference on Cooperative Design Visualization and Engineering (CDVE) [ticoCDVE] on 30^{th} April. This paper was accepted on 19^{th} May, and the final version was submitted on 10^{th} June.

The short-paper is entitled: A collaborative expandable framework for software end-users and programmers. This short-paper focus on the collaborative part of our framework, where the expert programmers create the parts that can be connected by end-users. At the time we first submitted this short-paper we simply had a work in progress, so that is why we didn't do a full paper. The short-paper is available in the Appendix at the Chapter A (p. 75).

5.3 FINAL CONSIDERATIONS

We didn't expect to create a new data-flow language, or a VPL or even a VDFPL. We believe these languages were the right influence for a solution that would solve our problem: We wanted to empower end-users with the ability to automate their smartphones with context-dependent tasks. We made a prototype that used several properties of these research areas, and we covered some problems of EUP and EUSE.

Our prototype is prepared to be expanded. The classes are loaded dynamically, tasks are serializable and blocks are easy to create. Every time the prototype was used by a new end-user, we saw something that could be improved. Some modifications were made thanks to that feedback, others were just saved in a todo list.

One item on our todo list was the rewrite rule system. End-users didn't like the amount of blocks needed to do a simple task. To do this rewrite rule system, we tried to use Prolog, but we were introducing some overhead that could be avoided, so we decided to try Scala. Scala mixes two paradigms: object-oriented and functional. This language already received some attention by the academic research and from the industry. Pattern matching and type inference were the

main reason why we decided to test Scala. We ended up only using pattern matching, since we stopped the Scala's development when we failed in the integration with our prototype. That made us forget this path, and we didn't try everything we wanted.

Sometimes we got lost on the features we could do. This thesis can suffer several ramifications, and we only explore a couple of them. One of those branches was a good experience but it didn't add value to our prototype. We left some loose ends, and maybe some of them could be finished when the Scala for Android gets more mature. Some improvements that we could do were registered, but we didn't have time to do all of them.

5.4 FUTURE WORK

We collected some ideas when we were working on this thesis. Some of them were implemented, others were delayed, and some of them were started but aren't complete.

5.4.1 Completing What is Done

We didn't complete some of the features that we started. This happened because we were always changing our priorities, and some features had to stop to give place to others. We grouped what it needs to be completed in this section.

Task and blocks management We prepared our prototype for tasks and blocks management, but the fact is that this is not visible. We could have a set of folders in the sdcard where the tasks and blocks would be saved[1]. Then, in the application, the end-user must be able to delete blocks he doesn't want as well as tasks. Additionally, it would be interesting that our system could suggest the deletion of some blocks. Tasks management should also have a module that checks what blocks are necessary for each new task and asks the end-user if he wants to download them.

Task creation improvements We have small things that need to be improved in the task creation. Removing a connection isn't intuitive and the lines connecting connectors can become confusing when they are more than five. A task should be, somehow, differentiated from a block (so users know they can edit it).

Improve block selection If we get about 30 blocks on the smartphone, it is not easy to find one. We need to improve the blocks selection in a medium or large number of blocks.

Use the Connection Types In this prototype, we only use the type of connection called "last value". We didn't conceive a way to insert more types in the user interface.

[1] Although each task is being saved in a file in a folder, the truth is that the end-user can't access that folder to transport a task from one device to another

Improve API When we were analyzing our solution with the state of the art in § 3.9 (p. 44) we realized that maybe we could improve our API. It would be interesting to try and improve our API to respect more rules from VDFPLS.

Finish the rewrite rule system Although we couldn't use what we have done Scala, we believe that this can be used someday. Finishing this system could be engaging, but it would be more interesting to think how we can integrate it with our prototype, without confusing the end-user. The end-user has to be able to see blocks that were grouped, so it should exist some kind of zoom. Zooming could be a way to define the abstraction level the end-user wants.

5.4.2 Exploring new Solutions

There are features that were never started because we knew that we needed some time to show something from this features. We choosed some of them that we thought that are interesting for future work.

A platform for pc/tablet to create tasks for the smartphone With more space in the screen, it is possible to conceive other ways to create a task. Since tasks are serializable, they could be created anywhere and they could be used on the smartphone.

A platform with tasks/blocks accessible by all end-users This is important because this is the channel that end-users and expert programmers will use to communicate. This platform should have means to request, select and download blocks. It is also important to build a trusty hierarchy between developers to prevent blocks that were doing more than expected from entering in the platform.

Study on how to avoid deadlocks on the smartphone One of the problems we thought is: An end-user can define a set of rules that result in a deadlock. Imagine the following example: If the smartphone is muted, then turn the sound on and If the smartphone has the sound on, then mute it. This is a simple example that would waste battery, but we can easily conceive examples that can turn off the screen and the user can never turn it on. If this happens the user can't open the application, so he has to turn off the smartphone and then turn it on again.

Create a way to represent connectors that can be connected through the use of implicits Although we didn't test implicits we know that we were going to have a problem with them. The way we represent types now in our prototype (with colors and with a message at the bottom) would become confusing with implicits. If the end-user can only connect blue outputs with blue inputs why he can connect, on a specific case, a blue with a red?

Appendices

Appendix A

Accepted short-paper

The following paper was submitted for the 9th conference on Cooperative Design Visualization and Engineering (CDVE) on 30^{th} April. This paper was accepted in 19^{th} May and the final version was submitted at 10^{th} June.

A collaborative expandable framework for software end-users and programmers

Tiago Almeida[1], Hugo Sereno Ferreira[1,2], and Tiago Boldt Sousa[1,2]
{tiago.silva.almeida, hugo.sereno, tiago.boldt}@fe.up.pt

[1] Department of Informatics Engineering, Faculty of Engineering, University of Porto
[2] INESC Technology and Science (formaly INESC Porto)

Abstract. The quantity and complexity that end-users are increasingly demanding from their applications and devices makes it impractical for a software developer to "foresee" every possible combination and explore every valid alternative. One solution is to empower end-users with tools that allows them to explore their necessities in a collaborative framework, where novices and experts can co-exist and share. We believe that such a tool could not only reduce the number of "small", specific-tailored applications, but also foster discovery and experimentation.

Keywords: Component-based Programming, End-user Programming, Cooperative Programming

1 Empowering end-users

End-users are all users who ultimately use software. The word *end-user* is someties mixed with the term *programming*, and when this happens, it can result in some misleading concepts. In this paper, we see an *end-user programmer* as someone who will program software for himself. Although, in principle, he can be an expert programmer (with a different experience background), the intended meaning of the term *end-user programmer* assumes the worst-case scenario: an end-user with no experience in programming, and a basic knowledge of the environment he is working with.

Why do we care about providing such development tools for end-users? Because end-users grow every year. In 2012 it is expected 90M end-users to be using newly developed software [1], contrasting with 3M expert programmers. But software, which is created for a specific group of people, can't answer (*a priori*) every specific end-user need. We thus aim to provide sufficient adaptability in software to allow its programming by an end-user [2].

On top of end-user programming (EUP), there is end-user software engineering (EUSE) [3]. It is the goal of the former to provide solutions for end-users to do some kind[3] of programming. EUSE, on the other hand, aims to empower those tools with deeper engineering concepts such as *testing* or *reuse*, so end-users can avoid errors and improve their productivity.

[3] "Kind" as in "basic".

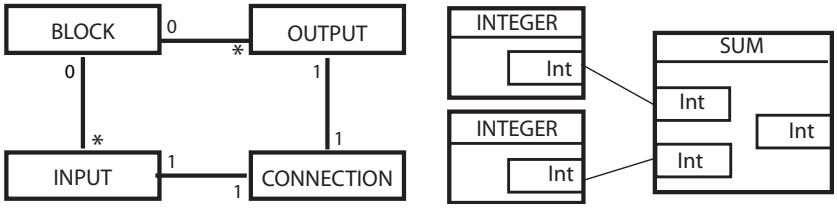
2 End-user programming

Although EUP and EUSE have different focus, some problems are convergent. Regarding software as inter-connected *pieces*, provides us a metaphor to understand the problems faced by end-users. Ko et el. [4] classified such end-user programming problems, by referring to the set of thoughts users had when they learning Visual Basic .NET. He then translated those problems to *thinking in pieces* instead of *thinking in software*: (a) Design, *I know what I want to do, but how?* (b) Selection, *What pieces can I use?* (c) Use, *I think this piece will do what I want, but how can I use it?* (d) Coordination, *How can I connect this set of pieces?* (e) Understanding, *This piece didn't do what I was expecting*, and (f) Information, *How can I see what my piece is doing?*.

Ko et el. also made a summary of the EUSE problems [3]. From those problems we will only focus in providing answers for Reuse — *How can I reuse what I already did and others did?* — and Debug — *How can I debug my pieces?*.

Current EUP solutions already cover many different areas, such as *Blender's* Composite Nodes for 3D modeling, *App Inventor* for mobile development, and *Kodu* for games.

3 A strongly typed block-based cooperative solution



(a) UML scratch (b) Example

Fig. 1: 1a) A block represented using UML; 1b) Representation to the end-user.

What we propose is a framework for both end-users and expert programmers to develop and connect pieces. We call those pieces Blocks.

When we conceived this idea we suspected that someone would already have thought in that. The truth is that a block with inputs and outputs is a common representation between software engineering. Black box testing, per example, uses this metaphor. This metaphor was also described by Zin [5] in a recent paper. The main idea is: We have a block, with a set of inputs and it returns outputs. These outputs can be connected with inputs, and both outputs and inputs have a type. Connections between blocks create another block.

A block can be represented using UML (fig 1a) and can be shown to the end-user using a box with inputs on the left and outputs on the right. Figure 1b shows that a block can connect its outputs with inputs from other block. In figure 1b we are summing two integer and producing a new integer.

We would like to separate the concepts of "conception" and "connection" of block, and provide a supply chain metaphor to our framework. Therefore, expert programmers would have tools to "concept" blocks for our system (supply), and end-users would have tools to "connect" these blocks (chain). A set of blocks is a block, however, users can name a set of blocks connected as a task. A task can do things like: "If the weather block has rain as output then activate the block that creates an alarm at 8h45". A task can be shared with others end-users and can be composed creating others tasks. All block management, connection, abstraction and information propagation on blocks would be controlled by our framework using a data-flow [6] approach. End-users and expert programmers will share a block metaphor and the same mechanism for information: data-flow.

4 Experience

We saw in the smartphone industry a good chance to test our concept. Smartphone users increase every day, and they already surpassed PCs in sales [7]. Another interesting thing in smartphones is that they provide a set of sensors that can be used to program simple tasks like: If I entered the campus turn my wifi connection on. However, end-users usually need an application for each task. Although this tendency is changing with applications like Tasker or even AutomateIt it still lacks of concepts like reuse and flexible design. We decided to develop a framework for Android, so we started with Java. We created a core that provides an API for expert programmers that can be used for every platform, not only Android. In fact, we tested this core using a PC implementation.

When the core was working and separated we started to think in GUI, and we realized that representing all blocks in a smartphone would be chaotic due to the small screen size. We couldn't conceive another representation, but one thing was for sure: We needed a way to group blocks. We then thought in a rewrite rule system, that could provide an expert programmer a way to specify automatic rules to group blocks. For this rewrite system, we tried to use Prolog, with an engine in Java. Soon we realized we were introducing an overhead in our system. We had to translate our information to facts, in Prolog, and then convert the results in Prolog to data, in Java. We decided to try Scala.

Scala is a high-level language defined over the JVM. It has an object oriented and a functional rib and it comes with a rich set of knowledge and new concepts [8]. It was hard to start working with Scala, but as soon as we started to go deeper in the Scala language, we found some advantages over Java. First, it is possible to provide an API that can come with in-built operators. Scala uses methods instead of operators and allows us to define methods such as: Block + Block. This provides whole new opportunities to simplify an API for expert programmers.

Also, this language provides a powerful tool named pattern matching that is usefull for our rewrite rule system, and some workflow engines (like Akka).

5 Future work

We would like to complete the change to Scala, and create our rewrite rule system that should support rules like: if two blocks called *integer provider* are connected with a block called *sum* (like shown on 1b) then create a replacement block called *GroupedSum*. This rule would be represented with something like `2xInteger Provider + Sum - GroupedSum`. Rules will only create visualization blocks. In fact, we will have the same blocks, but they are encapsulated to help users navigating through a complex set of blocks. However, users can and should see all blocks if they want to. Next we are going to develop a prototype for Android. This phase will be tricky. We need to study ways to represent the block concept in the smartphone that don't limit the flexibility. Also, we plan to implement a sharing platform, providing colaborative creation and sharing of blocks.

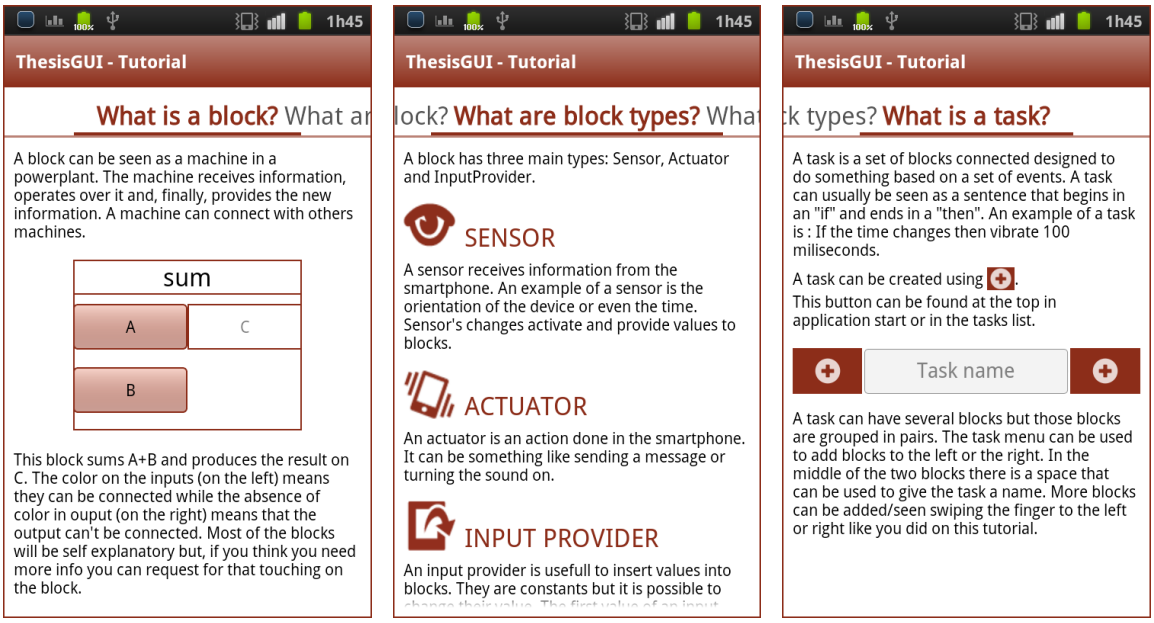
After all phases are completed, we want to test both our end-user as well as expert programmers frameworks. Both expert programmers and end-users will test the application. In this test, we will collect data directly with interviews. Next we will ask for expert programmers to develop a block and see the difficulties they will find. With this test, we intend to get answers for: Do we provide sufficient debug options? Our API is intuitive? Then we will ask them if they consider the concepts from the API are coherent with the main application. After those tests, we will analyze the data, fix some bugs and test again.

References

1. Scaffidi, C., Shaw, M., Myers, B.: Estimating the numbers of end users and end user programmers. In: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE Computer Society (2005) 207–214
2. Ferreira, H.S.: Adaptive object-modeling: Patterns, tools and applications (2010)
3. Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S.: The state of the art in end-user software engineering. ACM Comput. Surv. **43** (April 2011) 21:1–21:44
4. Ko, A.J., Myers, B.A., Aung, H.H.: Six learning barriers in end-user programming systems. In: Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing, IEEE Computer Society (2004) 199–206
5. Zin, A.: Block-Based Approach for End-User Software Development. Asian Journal of Information Technology **10**(6) (2011) 249–258
6. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. **36** (March 2004) 1–34
7. McKendrick, J.: More smartphones than pcs sold in 2011 `http://www.smartplanet.com/blog/business-brains/milestone-more-smartphones-than-pcs-sold-in-2011/21828`.
8. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Aritma Inc (2008)

Appendix B

Tutorial



(a) What is a block? (b) What are block types? (c) What is a task?

Figure B.1: The prototype tutorial done for end-users so they could have all the bases needed to use the prototype. Transition between each tutorial can be done through the swiping gesture.

Nomenclature

Activity An activity, in Android, is a screen with a user interface.

Actuator Block An actuator block (often abbreviated to actuator), is a type of block that executes an action on a smartphone (§ 24 (p. 27)).

API Application Programming Interface.

Block A block is an abstraction that has inputs, does something with those inputs and then produces outputs (§ 3.2.1 (p. 26)) .

Broadcast When we refer to broadcast we are referring to Android's broadcast. A broadcast, in Android, is a message emitted by the system to all applications.

Case classes A case class is a class that has some work done by the compiler like: a factory method with the name of the class so we can create classes without using the new keyword.

Connector A connector can be an input or an output (§ 3.2.2 (p. 28))..

Data-flow language Data-flow languages are based on the notion of data flowing from one function to another. This kind of language have five properties: Freedom from side-effects; Single assignment rule; Locality of effect; Data dependencies equivalent to scheduling andLack of history sensitivity in procedures (§ 2.2 (p. 14)) .

Debug The process used to locate and fix bugs.

DSL Domain-Specific Language .

DVM Dalvik Virtual Machine .

End-user In software engineering, it refers to an abstraction of the group of persons who will ultimately operate a piece of software, i.e., the expected user or target-user.

End-user programming "The practice by which end users write computer programs to satisfy a specific need, but programming is not their primary job function" [LSBM09] .

End-user software engineering End-user software engineering tries to "find ways to incorporate software engineering activities into users' existing work-flow, without requiring people to substantially change the nature of their work or their priorities" [KAB+11].

EUP End-User Programming [KAB+11].

EUSE End-User Software Engineering [KAB+11].

Facts Facts, in Prolog, are rules that always return true.

FIFO First In First Out.

GUI Graphical User Interface.

IDE Acronym for Integrated Development Environment.

Implicits Useful to convert one type to another depending on the context.

Input An input is the entrance where data is received. Usually an input is associated with a block.

Input Provider Block An input provider block (often abbreviated to input provider), is a type of block used to receive information from the end-user (§ 24 (p. 28)).

Intent An intent is usually related with an operation. A SMS intent can have, for instance, the number that send it and the content [Int].

JVM Java Virtual Machine .

LIFO Last In Last Out .

Output An output is the exit where data is send. Usually an output is associated with a block.

Pattern Matching Similar to the switch in Java, but it is more powerfull than a switch because we can compare all sort of objects.

Prolog A declarative language that is commonly used in artificial intelligence [Wik12].

Quick Action Menu A quick action menu is an Android pattern that is used for actions that are based in a context [dp].

Reuse The ability of using existing artifacts, or knowledge, to build or synthesize new solutions, or to apply existing solutions to different artifacts.

Rules Rules, in Prolog, are used to make conditional statements [Rul].

Scala A "scalable language" that blends "object-oriented and functional programming concepts in a statically typed language." [OSV08].

SDK Software-Development Kit .

Sensor Block A sensor block (often abbreviated to sensor), is a type of block that receives an event (§ 24 (p. 27)). .

Service A service is a component provided by the Android operating system that allows some long term processing in the background [Ser].

Swipe Swipe is a gesture that uses one finger. Swiping can be done from the left to the right or from right to the left. The gesture requires that the user touches the screen and moves the finger in an horizontal direction..

UML Unified Modeling Language .

VDFPL Visual Data-Flow Programming Languages.

Visual Programming Language A programming language that uses n-dimensional visual elements.

VPL Visual Programming Language.

XML eXtensible Markup Language .

References

[AA92] K. S. R. Anjaneyulu and John R. Anderson, *The advantages of data flow diagrams for beginning programming*, Proceedings of the Second International Conference on Intelligent Tutoring Systems (London, UK), Springer-Verlag, 1992, pp. 585–592. Cited on p. 15.

[Ack79] William B. Ackerman, *Data flow languages*, Managing Requirements Knowledge, International Workshop on **0** (1979), 1087. Cited on pp. 14 and 15.

[Akk] Akka, http://doc.akka.io/docs/akka/snapshot/scala/dataflow.html [Online; acessed 18-June-2012]. Cited on p. 61.

[AM90] Brad A. and Myers, *Taxonomies of visual programming and program visualization*, Journal of Visual Languages & Computing **1** (1990), no. 1, 97 – 123. Cited on pp. 17 and 18.

[Aut] AutomateIt, https://market.android.com/details?id=AutomateIt.mainPackage [Online; accessed 18-June-2012]. Cited on p. 5.

[BBB+95] M.M. Burnett, M.J. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee, *Scaling up visual programming languages*, Computer **28** (1995), no. 3, 45 –54. Cited on pp. 17 and 23.

[BCR04] Margaret Burnett, Curtis Cook, and Gregg Rothermel, *End-user software engineering*, Commun. ACM **47** (2004), 53–58. Cited on pp. 2 and 3.

[BD04] Marat Boshernitsan and Michael S. Downes, *Visual programming languages: a survey*, Tech. report, EECS Department, University of California, Berkeley, Dec. 2004. Cited on p. 18.

[BH95] Ed Baroth and Chris Hartsough, *Visual programming in the real world*, pp. 21–42, Manning Publications Co., Greenwich, CT, USA, 1995. Cited on pp. 4, 16, and 17.

[Bla06] A.F. Blackwell, *Psychological issues in end-user programming*, End User Development **9** (2006), 9–30. Cited on pp. 10 and 48.

[Bor86] A. Borning, *Defining constraints graphically*, SIGCHI Bull. **17** (1986), 137–143. Cited on p. 19.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler, *Making the future safe for the past: adding genericity to the java programming language*, SIGPLAN Not. **33** (1998), no. 10, 183–200. Cited on p. 51.

[Bur94] M Burnett, *A Classification System for Visual Programming Languages*, Sep. 1994, pp. 287–300. Cited on p. 18.

[Bur01] Margaret M. Burnett, *Visual programming*, John Wiley and Sons, Inc., 2001. Cited on pp. 4 and 17.

[Can] Canalys, http://www.canalys.com/ [Online; acessed 18-June-2012]. Cited on p. 1.

[CGP89] P.T. Cox, F.R. Giles, and T. Pietrzykowski, *Prograph: a step towards liberating programming from textual conditioning*, Visual Languages, 1989., IEEE Workshop on, oct 1989, pp. 150 –156. Cited on p. 19.

[CK00] Guanling Chen and David Kotz, *A survey of context-aware mobile computing research*, Tech. report, Hanover, NH, USA, 2000. Cited on p. 5.

[CK02] Jarinee Chattratichart and Jasna Kuljis, *Exploring the effect of control-flow and traversal direction on vpl usability for novices*, Journal of Visual Languages & Computing **13** (2002), no. 5, 471 – 500. Cited on pp. 4 and 17.

[Con] Android Developers: Context, http://developer.android.com/reference/android/content/Context.html [Online; accessed 18-June-2012]. Cited on p. 43.

[CXF+03] Li-Qun Chen, Xing Xie, Xin Fan, Wei-Ying Ma, Hong-Jiang Zhang, and He-Qin Zhou, *A visual attention model for adapting images on small displays*, Multimedia Systems **9** (2003), 353–364. Cited on p. 5.

[dap] Wolfram's data analysis platform, https://thestrangeloop.com/sessions/wolframs-data-analysis-platform [Online; acessed 18-June-2012]. Cited on p. 57.

[DK82] A.L. Davis and R.M. Keller, *Data flow program graphs*, Computer **15** (1982), no. 2, 26 – 41. Cited on pp. 14 and 20.

[DOR05] Enrico Denti, Andrea Omicini, and Alessandro Ricci, *Multi-paradigm java–prolog integration in tuprolog*, Science of Computer Programming **57** (2005), no. 2, 217 – 250. Cited on p. 55.

[dp] GoogleI/O 2010: Android UI design patterns, http://www.youtube.com/watch?v=M1ZBjlCRfz0#t=15m35s [Online; acessed 18-June-2012]. Cited on pp. 33 and 84.

[eoEUP] Google Tech Talk: The evolution of End User Programming, http://youtu.be/MxpjGZinies?t=53m27s [Online; acessed 18-June-2012]. Cited on p. 5.

[Era] The Java Tutorials: Type Erasure, http://docs.oracle.com/javase/tutorial/java/generics/erasure.html [Online; acessed 18-June-2012]. Cited on p. 51.

[Eur] European Spreadsheet Risks Interest Group, http://www.eusprig.org/stories.htm [Online; accessed 18-June-2012]. Cited on p. 11.

[fR] Android Developers: Designing for Responsiveness, http://developer.android.com/guide/practices/design/responsiveness.html [Online; acessed 18-June-2012. Cited on p. 5.

[Fra] Designing Your Own Framework, http://www.jamesbooth.com/designing_your_own_framework.htm [Online; accessed 18-June-2012]. Cited on p. 29.

[Gre95] Thomas Green, *Noddy's guide to ... visual programming*, The British Computer Society Human-Computer Group (1995). Cited on p. 4.

[Hil92] Daniel D. Hils, *Visual languages and computing survey: Data flow visual programming languages*, Journal of Visual Languages and Computing **3** (1992), no. 1, 69 – 101. Cited on p. 26.

[iBuNabi] Compositing in Blender using Nodes-a brief intro, http://members.dodo.com.au/iaina/BlenderNodePrimer.pdf [Online; acessed 18-June-2012]. Cited on p. 22.

[Int] Android Developers: Intent, http://developer.android.com/reference/android/content/Intent.html [Online; acessed 18-June-2012]. Cited on pp. 31 and 84.

[Jek] Logic Programming Jekejeke, http://www.jekejeke.ch/idatab/doclet/intr/en/docs/package.jsp [Online; acessed 18-June-2012]. Cited on p. 55.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar, *Advances in dataflow programming languages*, ACM Comput. Surv. **36** (2004), 1–34. Cited on pp. 14, 15, 16, and 21.

[KAB+11] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck, *The state of the art in end-user software engineering*, ACM Comput. Surv. **43** (2011), 21:1–21:44. Cited on pp. i, iii, 2, 3, 11, and 83.

[Kay93] Alan C. Kay, *The early history of smalltalk*, HOPL-II: The second ACM SIGPLAN conference on History of programming languages (New York, NY, USA), ACM, 1993, pp. 69–95. Cited on p. 1.

[KHA97] James D. Kiper, Elizabeth Howard, and Cuck Ames, *Criteria for evaluation of visual programming languages*, Journal of Visual Languages & Computing **8** (1997), no. 2, 175 – 192. Cited on p. 4.

[KMA04] A.J. Ko, B.A. Myers, and H.H. Aung, *Six learning barriers in end-user programming systems*, Visual Languages and Human Centric Computing, 2004 IEEE Symposium on, IEEE, 2004, pp. 199–206. Cited on pp. i, iii, 9, and 11.

[LH93] Ben Lee and A. R. Hurson, *Issues in dataflow computing*, Advances in computers (1993). Cited on p. 15.

[LSBM09] Grace Lewis, Dennis Smith, Len Bass, and Brad Myers, *Report of the workshop on software engineering foundations for end-user programming*, SIGSOFT Softw. Eng. Notes **34** (2009), no. 5, 51–54. Cited on pp. i, iii, 3, and 83.

[Mar97] Robert C. Martin, *Java and c++ a critical comparison*. Cited on p. 60.

[MKRT09] M. Marttila-Kontio, M. Ronkko, and P. Toivanen, *Visual data flow languages with action systems*, oct. 2009, pp. 589 –594. Cited on p. 19.

[MP00] M. Mosconi and M. Porta, *Iteration constructs in data-flow visual programming languages*, Computer Languages **26** (2000), no. 2–4, 67 – 104. Cited on p. 21.

[MSB11] Glenford J. Mayers, Corey Sandler, and Tom Badgett, *The art of software testing (3rd edition)*, John Wiley and Sons, Inc., Hoboken, N.J, 2011. Cited on p. 26.

[Mye] B. A. Myers, *Visual programming, programming by example, and program visualization: a taxonomy*, SIGCHI Bull. **17**, 59–66. Cited on p. 4.

[Nar93] Bonnie A. Nardi, *A small matter of programming: perspectives on end user computing*, MIT Press, Cambridge, MA, USA, 1993. Cited on p. 2.

[NPCN01] Raquel Navarro-Prieto and Jose J. Cañas, *Are visual programming languages better? the role of imagery in program comprehension*, Int. J. Hum.-Comput. Stud. **54** (2001), 799–829. Cited on pp. 4 and 17.

[Oa04] Martin Odersky and al., *An overview of the scala programming language*, Tech. Report IC/2004/64, EPFL Lausanne, Switzerland, 2004. Cited on pp. 51 and 57.

[OMG10] OMG, *Unified Modelling Language (UML)*, 2010, http://www.uml.org/ [Online; accessed 18-June-2012]. Cited on p. 16.

[oS] Twitter on Scala, http://www.artima.com/scalazine/articles/twitter_on_scala.html [Online; acessed 18-June-2012]. Cited on p. 57.

[oSEO] A Tour of Scala: Extractor Objects, http://www.scala-lang.org/node/112 [Online; acessed 18-June-2012]. Cited on p. 59.

[oSSC] A Tour of Scala: Sealed Classes, http://www.scala-lang.org/node/123 [Online; acessed cessed 18-June-2012]. Cited on p. 63.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners, *Programming in scala*, 2 ed., Aritma Inc, 2008. Cited on pp. 56, 57, 58, 60, and 84.

[OW97] Martin Odersky and Philip Wadler, *Pizza into java: translating theory into practice*, Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), POPL '97, ACM, 1997, pp. 146–159. Cited on p. 51.

[Pan08] Raymond R. Panko, *Spreadsheet errors: What we know. what we think we can do*, CoRR (2008). Cited on pp. 3, 11, and 12.

[Rec] Android Developers: Broadcast Receivers, http://developer.android.com/reference/android/content/BroadcastReceiver.html [Online; acessed 18-June-2012]. Cited on p. 35.

[Ref] Using Java Reflection, http://java.sun.com/developer/technicalArticles/ALT/Reflection/ [Online; acessed 18-June-2012]. Cited on p. 52.

[Rul] Prolog Tutorial Rules, http://www.doc.gold.ac.uk/~mas02gw/prolog_tutorial/prologpages/rules.html [Online; accessed 18-June-2012]. Cited on pp. 55 and 84.

[RZ10] Ahmad Rahmati and Lin Zhong, *A longitudinal study of non-voice mobile phone usage by teens from an underserved urban community*, CoRR **abs/1012.2832** (2010). Cited on pp. 1 and 4.

[San90] M. Santori, *An instrument that isn't really (laboratory virtual instrument engineering workbench)*, Spectrum, IEEE **27** (1990), no. 8, 36 –39. Cited on p. 19.

[SAW94] B. Schilit, N. Adams, and R. Want, *Context-aware computing applications*, Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on, dec. 1994, pp. 85 –90. Cited on p. 2.

[SBT] SBT, https://github.com/harrah/xsbt/wiki [Online; acessed 18-June-2012]. Cited on p. 65.

[Seg07] Judith Segal, *Some problems of professional end user developers*, Visual Languages and Human-Centric Computing (Los Alamitos, Ca, USA,) (Philip Cox and John Hosking, eds.), IEEE Computer Society Conference Publishing Services, Sep. 2007, pp. 111–118. Cited on p. 3.

[Ser] Android Developers: Service, http://developer.android.com/reference/android/app/Service.html [Online; acessed 18-June-2012]. Cited on pp. 31 and 84.

[Smu11] P. Smutny, *Visual programming for smartphones*, Carpathian Control Conference (ICCC), 2011 12th International, may 2011, pp. 358 –361. Cited on p. 5.

[SSM05] Christopher Scaffidi, Mary Shaw, and Brad Myers, *Estimating the numbers of end users and end user programmers*, Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (Washington, DC, USA), IEEE Computer Society, 2005, pp. 207–214. Cited on p. 2.

[stPSi] More smartphones than Pcs Sold in 2011, http://www.smartplanet.com/blog/business-brains/milestone-more-smartphones-than-pcs-sold-in-2011/21828 [Online; acessed 18-June-2012]. Cited on pp. i, 1, and 4.

[Tas] Tasker, http://tasker.dinglisch.net/ [Online; accessed 18-June-2012]. Cited on pp. 5 and 12.

[The] Language Theory, http://www.cs.man.ac.uk/~pjj/farrell/comp2.html#SYNTAX [Online; acessed 18-June-2012]. Cited on p. 62.

[ticoCDVE] The 9th international conference on Cooperative Design Visualization and Engineering, http://www.cdve.org/ [Online; acessed 18-June-2012]. Cited on p. 69.

[TMdHF11] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich, *Touchdevelop: programming cloud-connected mobile devices via touchscreen*, Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software (New York, NY, USA), ONWARD '11, ACM, 2011, pp. 49–60. Cited on p. 5.

[TOTK04] Sakari Tamminen, Antti Oulasvirta, Kalle Toiskallio, and Anu Kankainen, *Understanding mobile contexts*, Personal Ubiquitous Comput. **8** (2004), 135–143. Cited on p. 5.

[TT97] T.S.H. Teo and M. Tan, *Quantitative and qualitative errors in spreadsheet development*, System Sciences, 1997, Proceedings of the Thirtieth Hawaii International Conference on, vol. 3, Jan. 1997, pp. 149 – 155. Cited on p. 11.

[Use] ScalaCheck: UserGuide, https://github.com/rickynils/scalacheck/wiki/User-Guide [Online; acessed 18-June-2012]. Cited on pp. 61 and 64.

[Web] Blender Node Compositor Website, http://www.blender.org/development/release-logs/blender-242/blender-composite-nodes/ [Online; acessed 18-June-2012]. Cited on p. 22.

[Wik] Blender Node Compositor Wiki, http://wiki.blender.org/index.php/Doc:2.4/Manual/Composite_Nodes [Online; acessed 18-June-2012]. Cited on pp. 21 and 22.

[Wik12] Wikipedia, *Prolog — wikipedia, the free encyclopedia*, 2012, http://en.wikipedia.org/wiki/Prolog [Online; accessed 18-June-2012]. Cited on pp. 54 and 84.

[WNF06] Kirsten N. Whitley, Laura R. Novick, and Doug Fisher, *Evidence in favor of visual representation for the dataflow paradigm: An experiment testing labview's comprehensibility*, International Journal of Human-Computer Studies **64** (2006), no. 4, 281 – 303. Cited on pp. i, 4, and 17.

[WP] W-Prolog, http://waitaki.otago.ac.nz/~michael/wp/ [Online; acessed 18-June-2012]. Cited on p. 55.

[WP94] Paul G. Whiting and Robert S. V. Pascoe, *A history of data-flow languages*, IEEE Ann. Hist. Comput. **16** (1994), no. 4, 38–59. Cited on p. 61.

[Zin11] AM Zin, *Block-based approach for end-user software development*, Asian Journal of Information Technology **10** (2011), no. 6, 249–258. Cited on p. 26.