

Scuola di Ingegneria e Architettura
Corso di Laurea in Ingegneria e Scienze Informatiche

Progettazione di un ambiente di programmazione visuale block-based per ScaFi

Tesi di laurea in
(PROGRAMMAZIONE AD OGGETTI)

Relatore

Prof. Mirko Viroli

Candidato

Matteo Cerioni

Correlatore

Dott. Gianluca Aguzzi

Prof. Roberto Casadei

Quarta Sessione di Laurea
Anno Accademico 2020-2021

Sommario

Il paradigma *Aggregate Computing* semplifica lo sviluppo di applicazioni distribuite permettendo di utilizzare come punto di vista l'intero sistema distribuito anziché quello del singolo componente.

La toolchain *ScaFi* offre un linguaggio di programmazione basato su Scala per definire programmi che utilizzano i costrutti del paradigma *Aggregate Computing*.

La *programmazione visuale block-based* permette di comporre un programma connettendo graficamente tra loro dei blocchi tramite drag and drop.

Blockly è una libreria client-side utile per definire linguaggi di programmazione block-based.

Il progetto *Blockly2Scafi*, descritto in questa tesi, ha l'obiettivo di fornire tramite il linguaggio Scala.js un ambiente di programmazione visuale blockly-based, che consenta di definire programmi di un sottoinsieme delle funzionalità offerte dal linguaggio ScaFi, generandone codice testuale valido e ben leggibile.

Blockly2Scafi cattura gli elementi principali della programmazione ScaFi e ne semplifica l'approccio a utenti non esperti. L'editor visuale fornito da Blockly2Scafi viene integrato in una semplice e intuitiva *applicazione web* che offre all'utente la possibilità di sperimentare la costruzione di programmi ScaFi, visualizzando in tempo reale il codice generato.

Indice

Sommario	iii
Introduzione	1
1 Background su Aggregate Computing e ScaFi	3
1.1 Aggregate Computing	4
1.2 ScaFi (Scala Fields)	4
1.2.1 ScaFi-Web	5
1.2.2 L'esempio Channel in ScaFi	6
2 Background sulla programmazione visuale	7
2.1 Visual Programming Language	7
2.2 La programmazione visuale in ambito educativo ed accademico . . .	10
2.2.1 Scratch	10
2.2.2 Snap!	11
2.3 Blockly	12
3 Analisi e Requisiti	15
3.1 Requisiti funzionali	15
3.2 Requisiti non funzionali	16
3.3 Analisi	16
3.3.1 Scelta della tipologia del VPL	16
3.3.2 Scelta della libreria	16
3.3.3 L'uso di Blockly per il linguaggio ScaFi	17
3.3.4 JavaScript o Scala.js ?	17
4 Design e Implementazione	19
4.1 Definizione dei tipi di blocchi	19
4.1.1 Blockly Developer Tools	19
4.1.2 Caratteristiche dei blocchi	20
4.2 Workspace e toolbox	22

4.2.1	Workspace	22
4.2.2	Toolbox	23
4.3	Generatore di codice ScaFi	26
4.3.1	Importazione dei moduli ScaFi	26
4.3.2	Operator Precedence	28
4.4	L'interfaccia delle API in Scala.js	30
4.5	L'Applicazione web di Blockly2Scafi	31
5	Validazione	33
5.1	L'esempio Channel in Blockly2Scafi	33
5.2	Verifica dei requisiti	35
6	Conclusioni	37
6.1	Possibili sviluppi	37

Elenco delle figure

1.1	L'esempio channel eseguito tramite ScaFi-Web [10].	5
2.1	Kodu Games Lab: un VPL icon-based.	8
2.2	BlueJ: Un ambiente di sviluppo diagram-based orientato agli oggetti.	9
2.3	L'interfaccia web di Scratch 3.0.	11
2.4	L'editor di Blockly.	12
4.1	L'interfaccia Block Factory di Blockly Developer Tools.	20
4.2	Il workspace di Blockly2Scafi.	22
4.3	L'interfaccia dell'applicazione web di Blockly2Scafi.	31
5.1	L'esempio channel in Blockly2Scafi.	34

Elenco dei listati

4.1	La funzione JS di generazione del codice del blocco <code>mux</code>	26
4.2	La funzione JS di generazione del codice del blocco <code>aggregate_program</code>	27
4.3	La definizione in JS dell'ordine degli operatori	28
4.4	La funzione JS di generazione del codice del blocco <code>number_operation</code>	29
4.5	L'interfaccia delle API di Blockly2Scafi in <code>Scala.js</code>	30
4.6	Codice <code>Scala.js</code> dell'applicazione web	31
5.1	Codice ScaFi generato dall'esempio <code>channel</code> in Blockly2Scafi	34

Introduzione

Al giorno d'oggi esistono molti dispositivi computazionali in grado di connettersi tra loro, ma lo sviluppo di software in grado di coordinare tali dispositivi per perseguire collettivamente un unico obiettivo è ancora un'attività complessa.

ScaFi è un linguaggio che, implementando il paradigma Aggregate Computing, permette di semplificare lo sviluppo di applicazioni distribuite, permettendo di trattare il sistema distribuito come se fosse un'unica entità e senza doversi preoccupare della sincronizzazione dei singoli componenti.

ScaFi-Web è un'applicazione web che permette di visualizzare una simulazione di un sistema distribuito nel quale è possibile eseguire codice ScaFi e visualizzarne gli effetti nel sistema simulato.

L'obiettivo di ScaFi-Web è quello di semplificare l'accesso e l'apprendimento di ScaFi e dei costrutti dell'Aggregate Computing. Ciò si sposa con i valori e i vantaggi offerti dalla programmazione visuale.

L'idea dalla quale è nato questo progetto è di integrare nell'interfaccia di ScaFi-Web un *linguaggio di programmazione visuale basato sui blocchi*, in modo da semplificare la fase di apprendimento del linguaggio ScaFi.

Questa tesi ha l'obiettivo di realizzare un progetto preliminare dell'idea, sperimentandone la fattibilità tramite l'utilizzo della libreria Blockly, progettando un ambiente di sviluppo blockly-based per generare valido codice ScaFi per un sottoinsieme dei costrutti e delle funzionalità offerte dal linguaggio.

Il progetto è stato chiamato *Blockly2Scafi*.

Capitolo 1

Background su Aggregate Computing e ScaFi

Questo capitolo introduce i concetti del paradigma Aggregate Computing e di una sua implementazione, ScaFi.

Tradizionalmente, l'unità di base del calcolo computazionale è un singolo dispositivo dotato di una singola unità computazionale, successivamente sono nati i processori multi-core che, utilizzando le tecniche di programmazione parallela sono stati sfruttati per aumentare la potenza di calcolo dei dispositivi.

Oggi, grazie alla riduzione dei costi di produzione, sempre più dispositivi sono dotati di capacità computazionale e di connettività (smartphone, smartwatch, veicoli, dispositivi di assistenza persona, sensori intelligenti ecc.), di conseguenza, la necessità di sviluppare applicazioni distribuite è in aumento ed è quindi necessario sviluppare le tecniche di programmazione distribuita per semplificare tale processo.

Un *sistema distribuito* è formato da un gruppo decentralizzato di componenti interconnessi tra loro che comunicano esclusivamente tramite lo scambio di messaggi.

Ogni componente di un sistema distribuito ha una capacità computazionale indipendente ma, grazie alla capacità di comunicare tra loro, è possibile coordinare i singoli componenti per perseguire collettivamente un unico obiettivo.

La programmazione di sistemi distribuiti dal punto di vista del singolo componente risulta complessa in quanto ogni componente deve tenere conto della sincronizzazione e dello scambio di messaggi con gli altri.

1.1 Aggregate Computing

Il paradigma *Aggregate Computing* [22, 11] permette di definire programmi per sistemi distribuiti da un punto di vista complessivo (Computational Field).

L'Aggregate Computing permette di trattare il sistema distribuito come se fosse un'unica entità. Ciò semplifica la progettazione, la definizione e la manutenzione di programmi aggregati applicati a sistemi distribuiti che si evolvono nello spazio e nel tempo.

La base dell'Aggregate Computing è il *Field Calculus*, un set di costrutti che modellano la computazione e l'interazione di un grande numero di componenti spazialmente distribuiti.

Un esempio di applicazione distribuita che potrebbe usufruire del paradigma Aggregate Computing è un servizio per mantenere la sicurezza ed il controllo di ogni folla, ogni smartphone dei presenti, connettendosi con gli smartphone vicini, potrebbe comporre un sistema distribuito che permetterebbe una vista complessiva della folla ed indicare a ciascun dispositivo le istruzioni da seguire per mantenere la sicurezza [11].

1.2 ScaFi (Scala Fields)

ScaFi (Scala Fields) [13, 6] è una toolchain basata sul linguaggio di programmazione Scala che consente di definire ed eseguire programmi in un sistema distribuito simulato secondo il paradigma Aggregate Computing.

ScaFi Domain Specific Language (ScaFi DSL) è il linguaggio offerto da ScaFi per definire i programmi aggregati. ScaFi DSL è suddiviso in moduli e implementa i costrutti sia di base che di alto ordine del *field calculus*.

ScaFi implementa una Macchina Virtuale per eseguire i programmi aggregati definiti tramite il proprio linguaggio.

Inoltre, ScaFi permette di simulare un sistema distribuito per eseguire e controllare in locale il funzionamento del programma aggregato.

1.2.1 ScaFi-Web

ScaFi-Web [10, 7] è un'applicazione web che permette di scrivere ed eseguire programmi ScaFi in un comune web browser, semplificando l'accesso a ScaFi in quanto non necessita di alcuna installazione e preconfigurazione.

Nella fig. 1.1 viene mostrata l'interfaccia dell'applicazione, che include:

- Un editor interattivo per scrivere i programmi ScaFi;
- La rete dei dispositivi simulati connessi tra loro;
- La possibilità di modificare la configurazione della rete e dei sensori simulati.

Gli scenari d'uso di ScaFi-Web sono due:

- Nello scenario di apprendimento, ScaFi-Web semplifica l'approccio a ScaFi fornendo un tour guidato delle funzionalità offerte ed un elenco di esempi a difficoltà crescente.
- Nello scenario di prototipazione, ScaFi-Web permette di sviluppare e testare rapidamente programmi ScaFi, con il vantaggio di poter controllare graficamente ed in “tempo reale” il progresso dell'esecuzione nel simulatore.

Per poter funzionare in un web browser e per poter utilizzare ScaFi, ScaFi-Web viene compilato tramite Scala.js (Un cross-compiler da Scala a JavaScript).

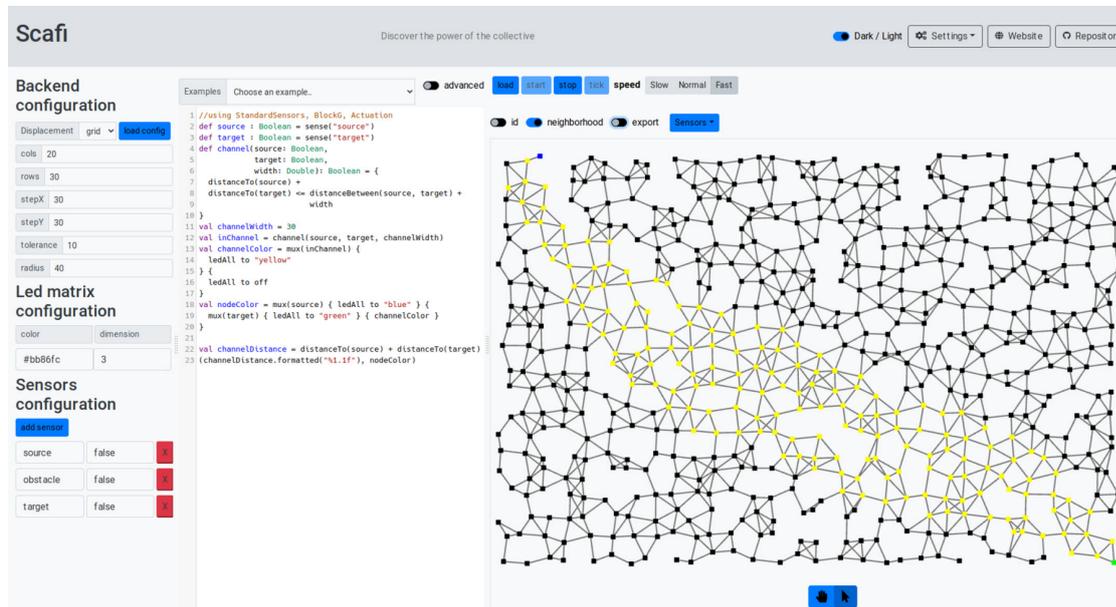


Figura 1.1: L'esempio channel eseguito tramite ScaFi-Web [10].

1.2.2 L'esempio Channel in ScaFi

L'esempio mostrato in fig. 1.1 è il Channel: un programma aggregato che stabilisce passo dopo passo un canale di connessione di distanza minima da un componente di partenza ad uno d'arrivo. Il canale si adatta dinamicamente ai cambiamenti del sistema distribuito: se i componenti si muovono nello spazio e nel tempo oppure se alcuni componenti si disconnettono e altri si riconnettono, il canale di connessione si ripristina automaticamente per garantire la connessione di distanza minima dei due componenti.

I vantaggi offerti dall'Aggregate Computing sono dimostrati dal fatto che l'algoritmo dell'esempio (scritto in ScaFi DSL) è definito in poche righe di codice ad un alto livello di astrazione e senza dover preoccuparsi della complessità di sviluppo sui singoli componenti.

Capitolo 2

Background sulla programmazione visuale

Questo capitolo mostra le tecniche ed i linguaggi di programmazione visuale più comuni e ne descrive i vantaggi rispetto alla programmazione testuale nel contesto educativo ed accademico.

La *programmazione visuale* [17] utilizza elementi grafici per rappresentare visivamente le regole e i costrutti della programmazione.

A differenza del tradizionale approccio testuale, la programmazione visuale permette ad un utente non esperto di trasformare le proprie idee in istruzioni, visualizzando a schermo una chiara rappresentazione del programma senza la necessità di conoscere le regole sintattiche, riducendo il rischio di incorrere in un errore ed abbassando il livello iniziale della curva di apprendimento [19].

2.1 Visual Programming Language

Un *linguaggio di programmazione visuale* (in inglese *Visual Programming Language - VPL*) è un qualsiasi linguaggio di programmazione che permette all'utente di manipolare il codice sottostante tramite l'utilizzo di strumenti grafici.

I VPL si possono generalizzare semanticamente in maniera parziale e sovrapposta tra le seguenti categorie [17]:

- I linguaggi **block-based** permettono all'utente di *comporre* un programma connettendo tra loro dei blocchi (un blocco è un elemento di programma predefinito) come se fossero dei pezzi di un puzzle, in modo tale da riuscire a comporre il flusso di istruzioni pensato. L'unione dei blocchi avviene trascinando questi ultimi, uno alla volta, da una lista di comandi predefinita ad un'area di sviluppo tramite drag and drop. Questo "incastro" che, ad occhio

può essere riconosciuto dalla forma, si realizza tramite la compatibilità logica tra gli elementi grafici rilevata dall'editor, il quale in caso contrario ne impedisce la connessione.

I linguaggi a blocchi eliminano la possibilità di commettere errori sintattici e favoriscono la creatività dell'utente in quanto non richiedono la conoscenza dei dettagli implementativi del linguaggio.

Ad esempio Scratch 2.2.1

- I linguaggi **icon-based** si basano sull'utilizzo delle icone e dei simboli grafici. Le icone elementari possono rappresentare oggetti, eventi o azioni che, combinate tra loro definiscono dei comportamenti di un programma. Questa tipologia di linguaggi è spesso utilizzata per permettere ad un utente inesperto di definire delle applicazioni basate su eventi ed azioni.

Ad esempio Kodu Games Lab fig. 2.1 [5].

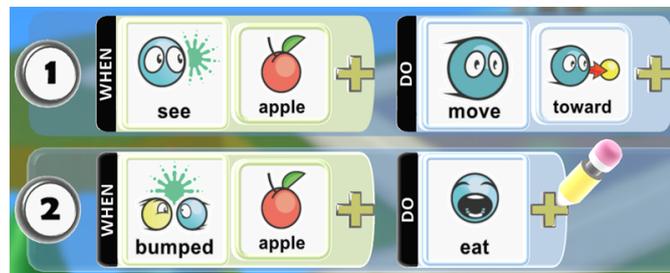


Figura 2.1: Kodu Games Lab: un VPL icon-based.

- I linguaggi **form-based** permettono all'utente di configurare dei comportamenti di un programma compilando delle form, per esempio selezionando degli eventi predefiniti e associandoli a delle azioni tramite dei menù a tendina. Alcuni linguaggi form-based sono solo visuali, altri invece offrono anche un approccio testuale per definire delle azioni o delle formule.

Ad esempio i comuni software di foglio di calcolo offrono la possibilità di definire formule in una cella in formato testuale, ma offrono anche la possibilità di utilizzare una form per selezionare graficamente le funzioni da utilizzare.

- I linguaggi **diagram-based** utilizzano i diagrammi per rappresentare il programma. Sono caratterizzati dalle connessioni grafiche degli oggetti tramite frecce o linee che ne rappresentano le relazioni.

I diagrammi di flusso possono essere utilizzati per rappresentare algoritmi, i diagrammi di stato per rappresentare i sistemi e i diagrammi delle classi per rappresentare il tipo di oggetti e le relazioni tra loro. Questi diagrammi vengono tradizionalmente utilizzati nelle fasi di analisi e progettazione del

software, ma in certi contesti è possibile utilizzarli anche nelle fasi implementative, migliorando la leggibilità e la comprensione del programma [12]. Ad esempio BlueJ fig. 2.2 [4].

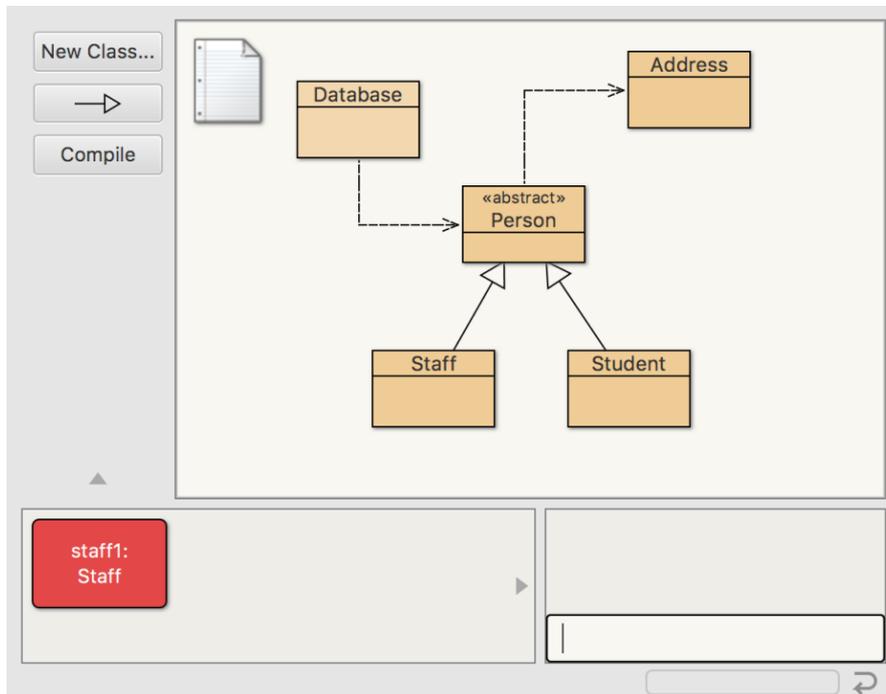


Figura 2.2: BlueJ: Un ambiente di sviluppo diagram-based orientato agli oggetti.

Nonostante la programmazione visuale risulti chiara e intuitiva nell'affrontare semplici problemi, all'aumentare della complessità alcuni VPL hanno la tendenza a perdere d'efficacia, in quanto visualizzando contemporaneamente molte informazioni viene ridotta l'attenzione sui singoli elementi [12].

Alcuni VPL, rispetto alla loro controparte testuale sono limitati, in quanto non sempre riescono a rendere accessibili tutte le possibili funzionalità mantenendo le proprie caratteristiche di semplicità e chiarezza. Una possibile soluzione adottata dai cosiddetti *linguaggi ibridi* [19] è quella di unire nella stessa interfaccia elementi grafici ad elementi testuali. Ad esempio, in un linguaggio block-based ibrido si potrebbe aggiungere la possibilità di personalizzare o di creare dei blocchi scrivendo delle formule matematiche o direttamente del codice. Un'altra possibilità offerta dall'approccio ibrido è di visualizzare in tempo reale il codice testuale generato, permettendo all'utente di familiarizzare con la sintassi e mantenendo la semplicità della programmazione visuale.

2.2 La programmazione visuale in ambito educativo ed accademico

La programmazione visuale applicata al settore educativo ed accademico non ha lo scopo di sostituire la programmazione testuale, ma ha l'obiettivo di focalizzare l'attenzione degli studenti verso i concetti della programmazione senza "distrarsi" con gli aspetti sintattici dei linguaggi testuali.

Per la fase di transizione dal linguaggio visuale al testuale vengono utilizzati i linguaggi ibridi.

Diversi studi [14, 15, 23] hanno evidenziato che, se gli studenti vengono introdotti alla programmazione tramite linguaggi visivi, apprenderanno meglio e faranno meno errori quando verranno introdotti all'uso dei linguaggi testuali. In particolare, la programmazione visuale aiuta gli studenti a superare alcune barriere iniziali della programmazione [16]:

- **Barriere di Selezione:** Quando il soggetto non riesce a determinare quale interfaccia di programmazione utilizzare per ottenere un determinato comportamento.
Ho capito cosa il programma debba fare, ma non capisco quale istruzione dovrei usare.
- **Barriere d'uso:** Quando il soggetto non riesce a capire come utilizzare e che effetti abbia una determinata interfaccia di programmazione.
Ho capito che istruzione devo utilizzare, ma non capisco come farla funzionare.
- **Barriere di Coordinazione:** Quando il soggetto non riesce a combinare tra loro interfacce di programmazione per ottenere comportamenti più avanzati.
Ho capito che istruzioni devo utilizzare, ma non capisco come farle funzionare insieme.

2.2.1 Scratch

Uno dei linguaggi di programmazione visuale block-based applicati al contesto educativo è *Scratch* [18, 8].

Scratch è utilizzato per introdurre i concetti della programmazione agli studenti con un età compresa tra 8 e 16 anni; usando un linguaggio naturale permette di costruire dei programmi orientati agli eventi per animare degli sprite (personaggi e oggetti creati dall'utente) in un ambiente a due dimensioni, stimolando la creatività per lo sviluppo di videogiochi 2D. Inoltre, Scratch è accessibile tramite un comune web browser e non richiede alcuna installazione per funzionare.

2.2. LA PROGRAMMAZIONE VISUALE IN AMBITO EDUCATIVO ED ACCADEMICO¹¹

L'interfaccia di Scratch, come si può vedere in fig. 2.3, è estremamente intuitiva: sulla sinistra è presente una lista di blocchi suddivisa per categorie; i blocchi possono essere personalizzati e trascinati nell'area centrale dove è possibile connetterli tra loro; a destra invece viene visualizzata la scena e lo stato attuale degli sprite. In questo caso al click sul personaggio, il gatto fa 10 passi in avanti e dice "Hello!".

Scratch mette a disposizione una serie di tutorial per imparare ad utilizzarlo da zero, inoltre contiene migliaia di progetti creati dagli utenti della community.

Scratch 3.0 è "Blockly-based" ovvero è basato sulla libreria Blockly [20].

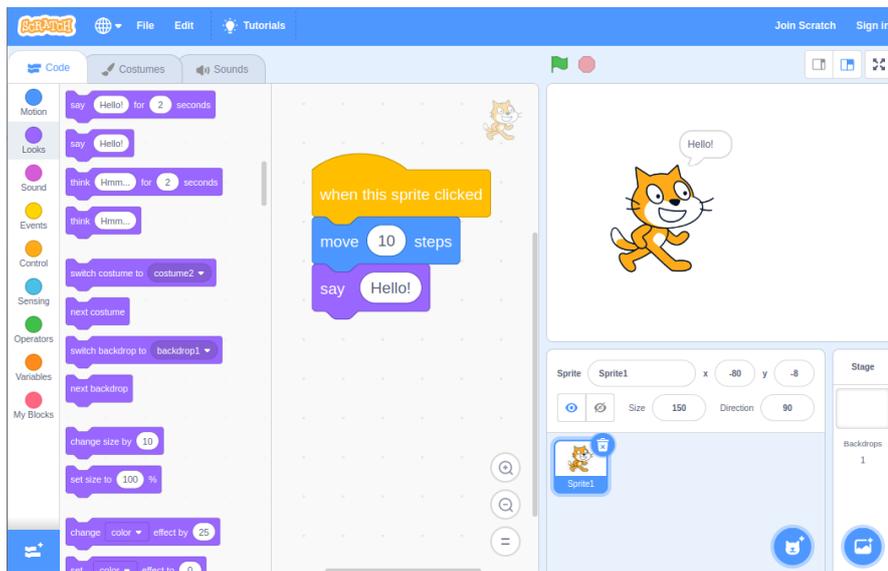


Figura 2.3: L'interfaccia web di Scratch 3.0.

2.2.2 Snap!

Snap! [9] è un linguaggio di programmazione visuale block-based a scopo educativo fortemente ispirato a Scratch. Questa similarità è possibile notarla nell'interfaccia del linguaggio visuale, nella gestione degli sprite e nella scena.

L'obiettivo di Snap! è di semplificare l'apprendimento e l'insegnamento dei costrutti avanzati della programmazione, infatti, le funzioni (blocchi) definite dall'utente possono essere utilizzate anche come valore per altri costrutti, permettendo di definire funzioni di alto ordine e quindi di introdurre agli studenti i concetti della programmazione funzionale. Anche le liste, come le funzioni, possono essere utilizzate come valori per altri costrutti; ciò permette di definire strutture dati avanzate come ad esempio liste di liste, alberi e mappe.

2.3 Blockly

Blockly [1, 21] è una libreria open source [2] utilizzata per creare editor di programmazione visuale block-based.

La libreria è scritta in JavaScript ed è quindi possibile utilizzarla sia all'interno di pagine web sia in applicazioni Android e iOS.

L'editor di Blockly è diviso in due sezioni: a sinistra si trova la toolbox, nella quale si trovano i blocchi selezionabili divisi per categorie; a destra si trova il workspace che consiste nello spazio di lavoro, nel quale è possibile connettere tra loro i blocchi. (Vedi fig. 2.4)

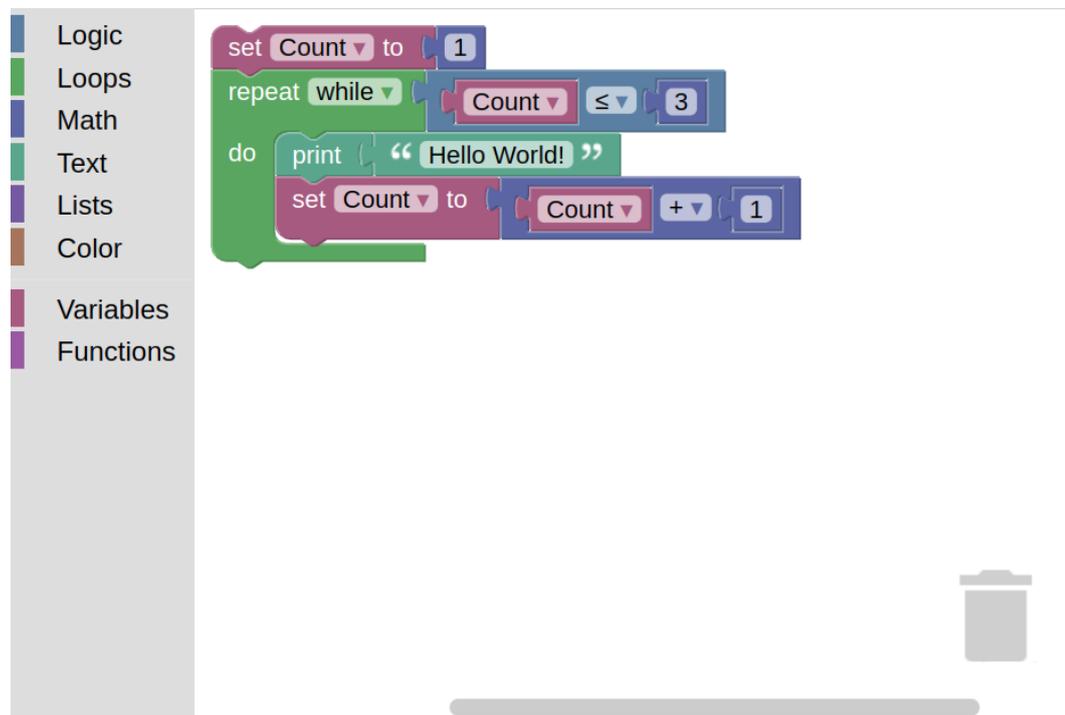


Figura 2.4: L'editor di Blockly.

La libreria comprende un set di blocchi preconfigurati che usano un vocabolario simile a quello dei linguaggi di programmazione testuale ma la libreria permette anche di creare dei propri blocchi personalizzati.

Oltre l'interfaccia dell'editor, Blockly include dei generatori di codice testuale per la traduzione dei blocchi preconfigurati per i linguaggi: JavaScript, Lua, PHP, Dart e Python. La libreria offre però, anche un framework per la costruzione di generatori di codice testuale personalizzati che rispettino la sintassi, l'indentazione e l'ordine delle istruzioni.

Di seguito alcune funzionalità offerte da Blockly:

- **Serializzazione:** fornisce la possibilità di serializzare il workspace, permettendo ad esempio di salvare lo stato attuale dei blocchi per caricarlo in un secondo momento.
- **Personalizzazione dei Temi:** offre la possibilità di configurare diversi temi stilistici. Ad esempio il tema scuro e il tema chiaro.
- **Internazionalizzazione:** mette a disposizione un sistema di traduzione; i blocchi default sono tradotti in oltre 40 lingue ed è anche supportata la scrittura da destra verso sinistra.
- **L'uso dei Menù contestuali:** consente di configurare azioni rapide raggiungibili tramite il click destro. Ad esempio si potrebbe aggiungere la funzionalità di copia di un gruppo di blocchi tramite click destro.
- **Controllo di tipo:** permette di definire le possibili connessioni dei blocchi.
- **Definizione di Variabili e funzioni:** supporta la possibilità di creare ed utilizzare variabili e funzioni.

Inoltre Blockly è *estensibile* pertanto permette di aggiungere una vasta gamma di plugin, sia supportati e garantiti dal team ufficiale di sviluppo di Blockly (First-party) [3], sia quelli creati dalla community (Third-Party).

Capitolo 3

Analisi e Requisiti

In questo capitolo vengono presentati i requisiti che il progetto deve soddisfare oltre ad un'analisi sulle tecnologie scelte per realizzarlo.

L'obiettivo di Blockly2Scafi è di definire un linguaggio di programmazione visuale che permetta di comporre graficamente programmi ScaFi per un sottoinsieme dei costrutti e delle funzionalità offerte dal linguaggio, con particolari attenzioni alle funzionalità necessarie per riprodurre l'esempio Channel.

Il VPL definito da Blockly2Scafi deve essere accessibile tramite un editor grafico integrabile in applicazioni web Scala.js.

Blockly2Scafi deve occuparsi della traduzione del codice dal linguaggio visuale al linguaggio testuale ScaFi.

Il progetto deve fornire anche una semplice applicazione web in Scala.js che utilizzi Blockly2Scafi per mostrare l'editor visuale ed il codice generato.

3.1 Requisiti funzionali

- R1** Blockly2Scafi deve funzionare nei moderni browser senza richiedere nessuna installazione o configurazione aggiuntiva.
- R2** Blockly2Scafi deve fornire un'API per permettere la sua integrazione in applicazioni web Scala.js.
- R3** Deve essere sviluppata una semplice applicazione web in Scala.js, la quale deve contenere l'editor visuale di Blockly2Scafi e deve visualizzarne il codice generato.
- R4** IL VPL di Blockly2Scafi deve offrire le funzionalità di ScaFi necessarie per comporre l'esempio Channel, in dettaglio:

- R4.1** Definire operazioni numeriche.
- R4.2** Definire comparazioni numeriche.
- R4.3** Definire e nominare variabili.
- R4.4** Ottenere i valori dai sensori.
- R4.5** Chiamare le funzioni di calcolo delle distanze.
- R4.6** Utilizzare il costrutto multiplexer.
- R4.7** Modificare il colore dei nodi.

R5 Il codice ScaFi generato deve essere valido; in dettaglio deve:

- R5.1** Essere sintatticamente corretto.
- R5.2** Essere ben formattato e ben indentato.
- R5.3** In caso di operazioni composite, per mantenere la leggibilità, deve evitare di utilizzare le parentesi quando non sono necessarie.

3.2 Requisiti non funzionali

- RNF1** Nell'applicazione web, la generazione del codice deve avvenire in tempo reale rispetto ai cambiamenti nell'editor dei blocchi.
- RNF2** Il generatore di codice deve importare automaticamente solo i moduli ScaFi richiesti dal programma.

3.3 Analisi

3.3.1 Scelta della tipologia del VPL

La scelta della tipologia del VPL è ricaduta in un linguaggio *block-based*, in quanto è la tipologia più adatta alla fase di apprendimento e di transizione da linguaggio visuale a testuale (sezione 2.2).

3.3.2 Scelta della libreria

La definizione di un VPL block-based e lo sviluppo dell'editor dei blocchi drag and drop di tale linguaggio sarebbero particolarmente complessi se sviluppati da zero senza l'ausilio di librerie o strumenti appositi.

Inizialmente è stato studiato *Snap!* (sezione 2.2.2) ma non si è rilevato adatto allo scopo di questo progetto sia perché non offre la possibilità di integrare l'editor

all'interno di altre applicazioni web, sia perché il suo funzionamento interno è legato all'uso degli sprite nella scena.

La scelta di tale libreria è infine ricaduta su *Blockly* (sezione 2.3), in quanto, Blockly nasce proprio come una libreria per lo sviluppo di VPL block-based, quindi è completamente personalizzabile, integrabile in applicazioni esterne e semplifica la fase di generazione di codice testuale.

3.3.3 L'uso di Blockly per il linguaggio ScaFi

Le caratteristiche e i costrutti di ScaFi non sono banali da mappare in un linguaggio visuale infatti: i programmi ScaFi fanno largo uso della programmazione funzionale e, nonostante il linguaggio si possa considerare type-safe, in certi casi alcune funzioni restituiscono un valore con un tipo generico (ad esempio `sense`).

Purtroppo di default la gestione delle variabili e delle funzioni offerta da Blockly non è type-safe e le funzioni definite dall'utente non possono essere utilizzate come valore. Fortunatamente però la capacità di personalizzazione di Blockly permette con qualche sforzo in più di definire i comportamenti delle variabili e delle funzioni da zero, rendendo quindi possibile l'integrazione in Blockly delle varie caratteristiche di ScaFi.

Il fattore chiave di un linguaggio di programmazione visuale a scopo di apprendimento deve rimanere la sua semplicità d'uso; alcune funzionalità avanzate della controparte testuale potrebbero essere limitate e semplificate nel linguaggio visuale.

3.3.4 JavaScript o Scala.js ?

Per semplificare la futura integrazione del progetto in ScaFi-Web, Blockly2Scafi deve fornire un'API per Scala.js in modo type-safe.

La libreria Blockly, ad oggi, non dispone di supporto a TypeScript, quindi, utilizzare Blockly in Scala.js in modo type-safe richiederebbe di riscrivere le interfacce e di mappare i tipi della libreria di Blockly.

La soluzione scelta è quella di scrivere in Scala.js in modo type-safe solo l'interfaccia delle API di Blockly2Scafi utili alla sua integrazione; il funzionamento interno del progetto invece, per poter utilizzare Blockly, viene scritto direttamente tramite il linguaggio JavaScript.

Capitolo 4

Design e Implementazione

Questo capitolo descrive la progettazione e i dettagli implementativi di Blockly2Scafi e dell'applicazione web.

Data la natura esplorativa del progetto, si è optato per la scelta del modello di sviluppo del software incrementale di prototipazione evolutiva.

Le fasi di design e di implementazione si sono spesso interscambiate, partendo da un primo prototipo che, passo dopo passo, è stato migliorato ed esteso cercando di sfruttare le funzionalità offerte dalla libreria Blockly e cercando di ottenere l'effetto desiderato dall'ambiente di programmazione visuale.

4.1 Definizione dei tipi di blocchi

Data la diversità dei costrutti del linguaggio ScaFi rispetto a quelli dei linguaggi di programmazione supportati da Blockly si è optato di non utilizzare alcun blocco della libreria di default di Blockly quindi la libreria utilizzata è completamente composta da blocchi personalizzati.

4.1.1 Blockly Developer Tools

Blockly Developer Tools è un'applicazione web che fornisce strumenti utili alla configurazione e alla personalizzazione di applicazioni Blockly.

In particolare lo strumento *Block Factory* mostrato in fig. 4.1 permette di definire i blocchi personalizzati. Lo strumento Block Factory usa a sua volta Blockly per permettere di configurare i blocchi, velocizzando e semplificando il processo, in quanto permette di evitare di scrivere manualmente il JSON di configurazione. Inoltre viene visualizzata un'anteprima aggiornata in tempo reale del blocco creato.

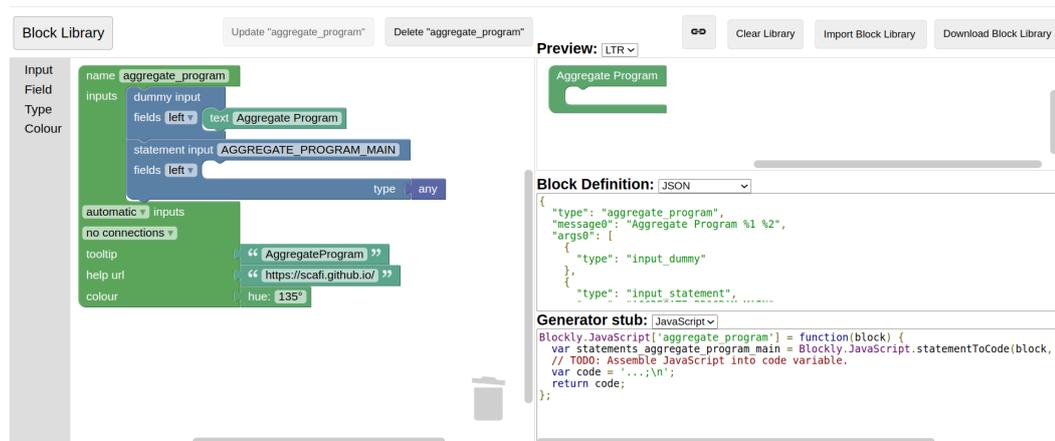


Figura 4.1: L'interfaccia Block Factory di Blockly Developer Tools.

4.1.2 Caratteristiche dei blocchi

Ogni blocco è caratterizzato principalmente dalla sua forma, che è definita dalle proprie connessioni di input e di output.

In questa sezione ne analizzeremo i dettagli mostrando esempi grafici.

Connessioni di output L'output di un blocco è definito dalle connessioni posizionate al di sopra, al di sotto e alla sinistra del blocco; ne esistono di 3 tipi:

Nessun output: sono i blocchi che non hanno nessuna connessione in output, concettualmente definiscono un blocco contenitore di altri blocchi.
Ad esempio il blocco di definizione della funzione aggregate program in figura.



Output di un valore: sono i blocchi che hanno una connessione verso sinistra, concettualmente restituiscono un valore al blocco al quale sono connessi.
Ad esempio il blocco di definizione di una stringa in figura.



Output di un'istruzione: sono i blocchi che hanno una connessione verso l'alto e verso il basso, quindi possono essere connessi consecutivamente e concettualmente definiscono un'istruzione.
Ad esempio il blocco di definizione di una variabile in figura.



Connessioni di input L'input di un blocco è definito dalle connessioni interne ed alla destra del blocco, a differenza delle connessioni di output, un blocco può avere più di una connessione di input; in dettaglio:

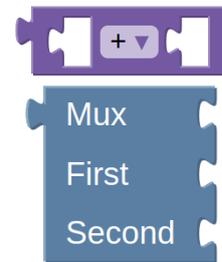
Caselle di input: sono le caselle di input nelle quali l'utente può scrivere o selezionare direttamente un valore, ce ne sono di diverse tipologie: input di testo, input numerico, menù a tendina, color picker ecc. Ad esempio in figura vengono mostrati:

- il blocco di inserimento di un numero intero.
- il blocco di inserimento di un valore booleano.



Input di valore: sono le connessioni di input che richiedono un blocco che restituisce in output un valore. Graficamente possono essere visualizzate in due modi: "inline" (connessioni interne al blocco) ed "external" (connessioni alla destra del blocco). Ad esempio in figura vengono mostrati:

- il blocco per definire operazioni matematiche, che contiene due input di valore "inline".
- il blocco del costrutto multiplexer, che contiene tre connessioni di input "external".



Input di istruzione: sono le connessioni di input che richiedono un blocco che restituisce in output un'istruzione. Ad esempio il blocco di definizione della funzione aggregate program in figura.



Connessioni type-safe Le connessioni per valore possono avere dei *vincoli sul tipo*, ad esempio prendendo in considerazione il blocco per definire operazioni matematiche, i due input dei valori hanno un vincolo che blocca tutte le connessioni che restituiscono un valore di tipo diverso da Integer o Double.

4.2 Workspace e toolbox

I due elementi che compongono l'editor di Blockly sono il *Workspace* (l'area dove i blocchi possono essere connessi tra loro) e la *Toolbox* (la barra laterale divisa per categoria dalla quale è possibile trascinare i nuovi blocchi), in questa sezione vedremo come sono stati configurati.

4.2.1 Workspace

Il *workspace* è stato configurato in modo da contenere un blocco di definizione di un programma aggregato chiamato `aggregate_program`.

Il blocco `aggregate_program` ha lo scopo di essere il contenitore del programma; rappresenta la funzione "main".

Il blocco `aggregate_program` è univoco, non è cancellabile e tutti i blocchi nella workspace che non sono all'interno di esso vengono ignorati dal generatore di codice e visualizzati in grigio come mostrato in fig. 4.2.

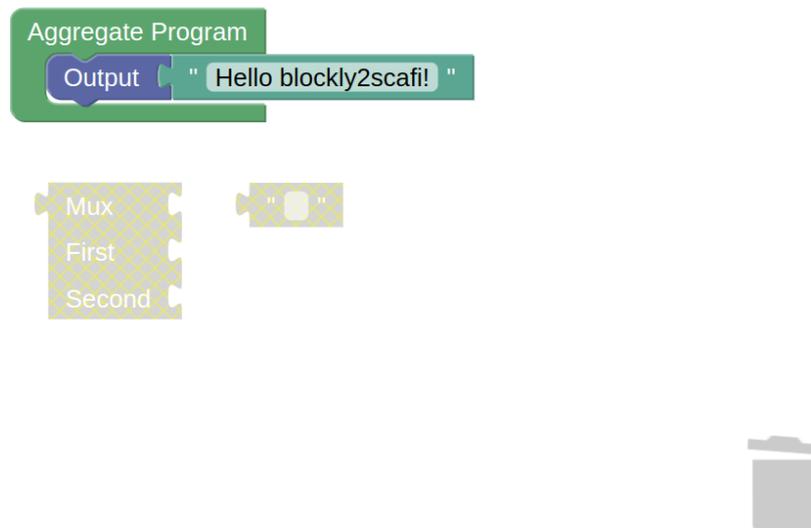


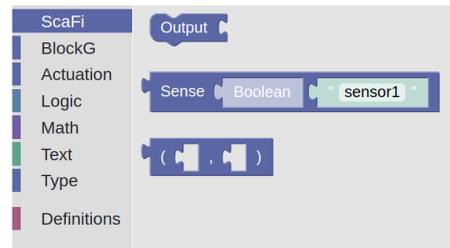
Figura 4.2: Il workspace di Blockly2Scaffi.

4.2.2 Toolbox

La *toolbox* è configurata per permettere di visualizzare e selezionare tutti i blocchi definitivi suddivisi per categoria, in dettaglio:

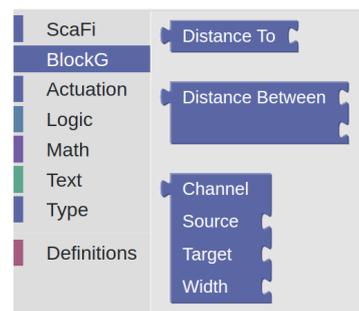
ScaFi:

- **output**, è il blocco per restituire in output un valore.
- **sense**, è il blocco per ottenere il valore di un sensore.
- **tupla**, è il blocco per definire una tupla di due valori.



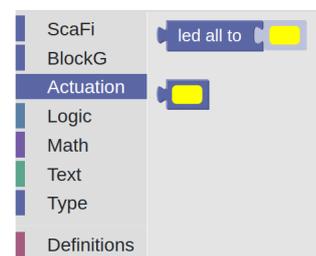
BlockG:

- **distance_to**, è il blocco per ottenere la distanza da un'origine.
- **distance_between**, è il blocco per ottenere la distanza da un'origine ad un obiettivo.
- **channel**, è il blocco per ottenere un canale da un'origine ad un obiettivo.



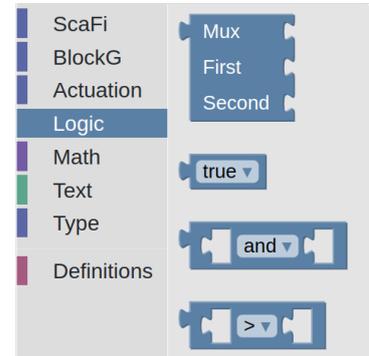
Actuation:

- **led_all_to**, è il blocco per cambiare il colore dei led.
- **color**, è il blocco per definire un colore, contenente un color picker.

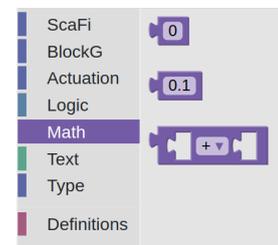


Logic:

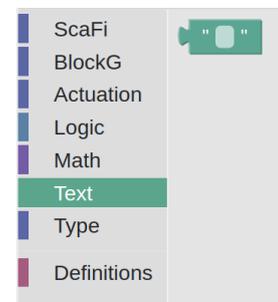
- `mux`, è il blocco per il costrutto multiplexer.
- `boolean`, è il blocco di un valore booleano.
- `boolean_operation`, è il blocco per definire un'operazione logica: `and` e `or`.
- `number_compare`, è il blocco per definire una comparazione numerica: `>` `>=` `==` `!=` `<=` `<`.

**Math:**

- `integer`, è il blocco di un valore intero.
- `double`, è il blocco di un valore reale.
- `number_operation`, è il blocco per definire un'operazione numerica: `+` `-` `*` `/` `%`.

**Text:**

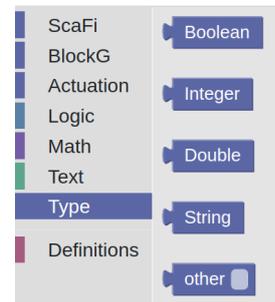
- `string`, è il blocco per definire una stringa.



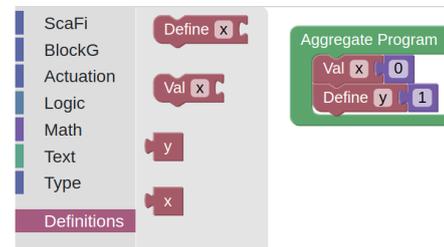
Type:

Questa categoria contiene i blocchi per assegnare il tipo di ritorno al blocco **sense**:

- i blocchi **type** preconfigurati per i tipi più comuni.
- il blocco **other_type** che permette di descrivere un qualsiasi altro tipo.

**Definitions:**

Questa è una categoria dinamica, in quanto innanzitutto, contiene i due blocchi di definizione di una variabile: **def** e **val**. Successivamente vengono aggiunti dinamicamente i blocchi **getter** delle variabili definite, in modo da poterli utilizzare come valore.



4.3 Generatore di codice ScaFi

Il *generatore di codice ScaFi* è il componente che ha il ruolo di “tradurre” il programma composto tramite l’editor Blockly in codice ScaFi valido e leggibile.

Il funzionamento interno del Code Generator è suddiviso per ogni tipo di blocco. Per ogni tipo di blocco è stata definita una funzione JavaScript che ne generi il codice testuale “assemblando” le stringhe di codice ricevute dalle proprie connessioni di input.

Se il blocco ha un output di istruzione, tale funzione deve restituire solo il codice generato, invece se il blocco ha un output di valore, la funzione deve restituire anche la priorità del codice generato rispetto agli altri, in modo tale che la composizione del codice utilizzi le parentesi per separare le espressioni solo quando sia necessario (ulteriori dettagli nella sottosezione 4.3.2).

Per esempio la funzione di generazione di codice del costrutto `mux` è presentata nel listato 4.1; tale funzione è suddivisa in due fasi: raccogliere gli input ed assemblare il codice.

Listato 4.1: La funzione JS di generazione del codice del blocco `mux`

```

1 scafiGenerator['mux'] = function (block) {
2   const condition = Blockly.ScaFi.valueToCode(block, 'CONDITION'
3     , scafiGenerator.ORDER_ATOMIC);
4   const firstBranch = Blockly.ScaFi.valueToCode(block, '
5     FIRST_BRANCH', scafiGenerator.ORDER_ATOMIC);
6   const secondBranch = Blockly.ScaFi.valueToCode(block, '
7     SECOND_BRANCH', scafiGenerator.ORDER_ATOMIC);
8
9   let code = 'mux(' + condition + '){\n';
10  code += scafiGenerator.prefixLines(firstBranch, scafiGenerator
11    .INDENT)+'\n';
12  code += '}{\n';
13  code += scafiGenerator.prefixLines(secondBranch,
14    scafiGenerator.INDENT)+'\n';
15  code += '}'
16
17  return [code, scafiGenerator.ORDER_ATOMIC];
18 }

```

4.3.1 Importazione dei moduli ScaFi

Alcune funzioni di ScaFi dipendono dall’importazione di determinati moduli, di conseguenza un’importante funzionalità che deve offrire il generatore di codice di ScaFi è quella di *importare automaticamente i moduli* ai quali i blocchi che si stanno utilizzando dipendono.

Tale responsabilità è assegnata alla funzione di generazione del codice del blocco `aggregate_program` (listato 4.2), la quale, tramite una mappa che ha come chiave il tipo di blocco e come valore l'elenco delle sue dipendenze, costruisce l'array di dipendenze dei blocchi utilizzati.

Listato 4.2: La funzione JS di generazione del codice del blocco `aggregate_program`

```

1  scafiGenerator[ 'aggregate_program' ] = function (block) {
2    const import_map = {
3      "distance_to" : [ "StandardSensors", "BlockG" ],
4      "distance_between" : [ "StandardSensors", "BlockG" ],
5      "channel" : [ "StandardSensors", "BlockG" ],
6      "led_all_to" : "Actuation",
7    }
8
9    let importArray = [];
10   let import_code = "";
11
12   const workspace = block.workspace;
13   const allBlocks = workspace.getAllBlocks();
14   for(const block of allBlocks){
15     if(block.type in import_map){
16       let modules = import_map[block.type];
17       if(!Array.isArray(modules)){
18         modules = [modules];
19       }
20       for(const module of modules){
21         if(!importArray.includes(module)){
22           importArray.push(module);
23         }
24       }
25     }
26   }
27   if(importArray.length){
28     import_code = "//using "+importArray.join(", ")+"\n";
29   }
30
31   const otherCode = Blockly.ScaFi.blockToCode(block.
32     getInputTargetBlock("AGGREGATEPROGRAMMAIN"));
33
34   return import_code + otherCode;
35 }

```

4.3.2 Operator Precedence

Un fattore importante che deve gestire il generatore di codice è quello di applicare l'ordine delle operazioni per mantenere il codice leggibile.

Quando si definiscono operazioni composite come le espressioni numeriche o logiche, è importante che le parentesi vengano utilizzate solo dove sia necessario.

Ad esempio per l'espressione `(A || B || C) && D` è necessaria una sola parentesi in quanto l'operatore `&&` ha una priorità maggiore rispetto all'operatore `||`; se il generatore di codice non gestisse tali priorità, si otterrebbe l'espressione `((A || B) || C) && D` che risulterebbe poco leggibile.

Ciò è possibile definendo una serie di priorità sugli operatori del linguaggio (listato 4.3) che verranno utilizzate nella funzione di generazione del codice, come ad esempio in quella dell'operazione numerica (listato 4.4).

Listato 4.3: La definizione in JS dell'ordine degli operatori

```
1 scafiGenerator.ORDER_ATOMIC = 0;
2 scafiGenerator.ORDER_FUNCTION_CALL = 2;           // ()
3 scafiGenerator.ORDER_MULTIPLICATION = 5.1;       // *
4 scafiGenerator.ORDER_DIVISION = 5.2;             // /
5 scafiGenerator.ORDER_MODULUS = 5.3;              // %
6 scafiGenerator.ORDER_SUBTRACTION = 6.1;          // -
7 scafiGenerator.ORDER_ADDITION = 6.2;             // +
8 scafiGenerator.ORDER_RELATIONAL = 8;             // < <= > >=
9 scafiGenerator.ORDER_EQUALITY = 12;              // == !=
10 scafiGenerator.ORDER_LOGICAL_AND = 13;          // &&
11 scafiGenerator.ORDER_LOGICAL_OR = 14;           // ||
12 scafiGenerator.ORDER_ASSIGNMENT = 20;           // =
13 scafiGenerator.ORDER_NONE = 99;
```

Listato 4.4: La funzione JS di generazione del codice del blocco number_operation

```
1 scafiGenerator['number_operation'] = function (block) {
2   const operation = block.getFieldValue("OPERATOR");
3
4   let order;
5   let operator;
6   if(operation === 'ADDITION'){
7     operator = ' + ';
8     order = scafiGenerator.ORDER_ADDITION;
9   }else if(operation === 'SUBTRACTION'){
10    operator = ' - ';
11    order = scafiGenerator.ORDER_SUBTRACTION;
12  }else if(operation === 'MULTIPLICATION'){
13    operator = ' * ';
14    order = scafiGenerator.ORDER_MULTIPLICATION;
15  }else if(operation === 'DIVISION'){
16    operator = ' / ';
17    order = scafiGenerator.ORDER_DIVISION;
18  }else{ //MODULUS
19    operator = ' % ';
20    order = scafiGenerator.ORDER_MODULUS;
21  }
22
23  const first = Blockly.ScaFi.valueToCode(block, 'FIRST', order)
24    ;
25  const second = Blockly.ScaFi.valueToCode(block, 'SECOND',
26    order);
27  const code = first + operator + second;
28  return [code, order];
29 }
```

4.4 L'interfaccia delle API in Scala.js

L'interfaccia delle API di Blockly2Scafi in Scala.js è composta da tre entità (listato 4.5):

- L'oggetto `Blockly` contiene la funzione `createBlockly2ScafiWorkspace` per inizializzare il workspace in un dato elemento del DOM; inoltre contiene il riferimento al generatore di codice `ScaFi`.
- Il trait `Workspace` offre la possibilità di aggiungere un listener degli eventi tramite la funzione `addChangeListener`.
- Il trait `ScaFi` rappresenta il generatore di codice; offre la possibilità di ottenere il codice generato a partire dal workspace tramite la funzione `workspaceToCode`.

Un esempio di integrazione delle API è mostrato nel listato 4.6.

Listato 4.5: L'interfaccia delle API di Blockly2Scafi in Scala.js

```
1 @js.native
2 trait ScaFi extends js.Object { //ScaFi custom code generator
3   def workspaceToCode(workspace: Workspace): String = js.native
4 }
5
6 @js.native
7 trait Workspace extends js.Object {
8   def addChangeListener(function: js.Function): Unit
9 }
10
11 @js.native
12 @JSGlobal
13 object Blockly extends js.Object {
14   def createBlockly2ScafiWorkspace(elt: Element): Workspace = js.
15     native
16
17   def ScaFi: ScaFi = js.native
18 }
```

4.5 L'Applicazione web di Blockly2Scafi

Per visualizzare e testare il progetto è stata implementata una semplice applicazione web scritta in Scala.js e che integra Blockly2Scafi (fig. 4.3).

Il layout riporta fianco a fianco l'editor di Blockly2Scafi ed una sezione dove viene visualizzato il codice generato in real-time.

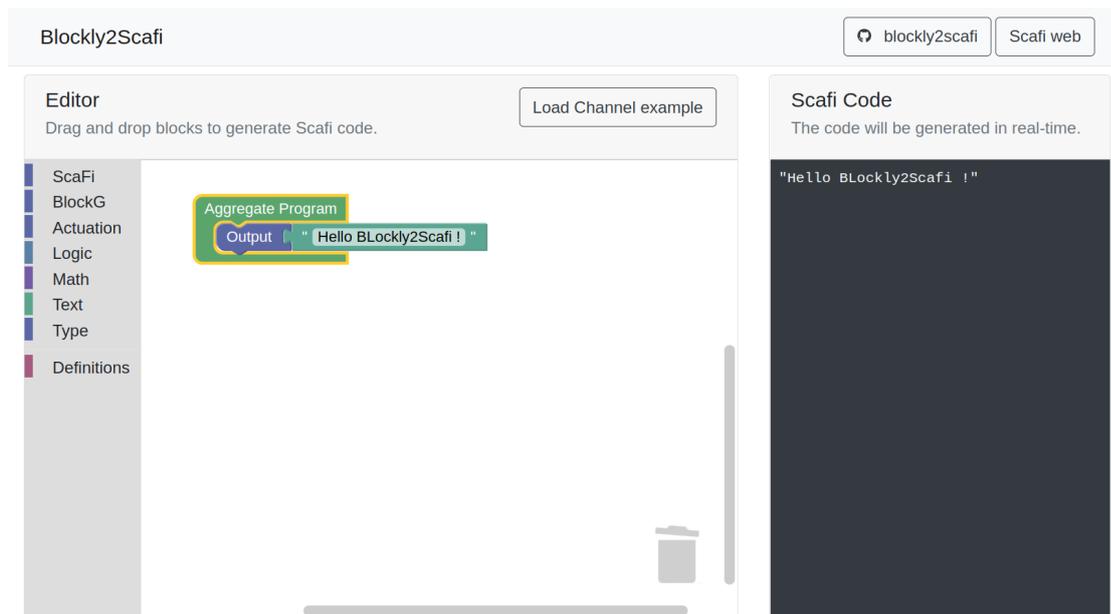


Figura 4.3: L'interfaccia dell'applicazione web di Blockly2Scafi.

Il codice in Scala.js dell'applicazione è presentato nel listato 4.6, nel quale viene inizializzato Blockly2Scafi ed aggiunto un evento al workspace in modo che ad ogni modifica venga rigenerato e visualizzato il codice ScaFi.

Listato 4.6: Codice Scala.js dell'applicazione web

```

1 object App {
2   def main(args: Array[String]): Unit = {
3     val BlocklyEditorId = "blockly-editor"
4     val GeneratedCodeId = "generated-code"
5
6     val blocklyEditorElement = document.getElementById(
7       BlocklyEditorId)
8     val generatedCodeElement = document.getElementById(
9       GeneratedCodeId)
10    val workspace = Blockly.createBlockly2ScafiWorkspace(
11      blocklyEditorElement)
12
13    workspace.addChangeListener(() => {
14      generatedCodeElement.textContent = Blockly.ScaFi.
15        workspaceToCode(workspace)
16    })
17  }
18 }

```


Capitolo 5

Validazione

Questo capitolo mostra i risultati ottenuti dal progetto e verifica che i requisiti descritti nella fase di analisi siano stati rispettati.

Data la complessità di implementazione di test automatici per il linguaggio visivo, la verifica del corretto funzionamento delle varie funzionalità è stata effettuata tramite le tecniche di collaudo white box e black box.

5.1 L'esempio Channel in Blockly2Scafi

Un importante obiettivo conseguito dal progetto è di permettere la definizione dell'esempio Channel (1.2.2) tramite Blockly2Scafi, in fig. 5.1 viene visualizzata la composizione dei blocchi necessaria ad ottenere la generazione del codice mostrato nel listato 5.1. Il codice generato è stato infine testato con esito positivo in ScaFi-Web.

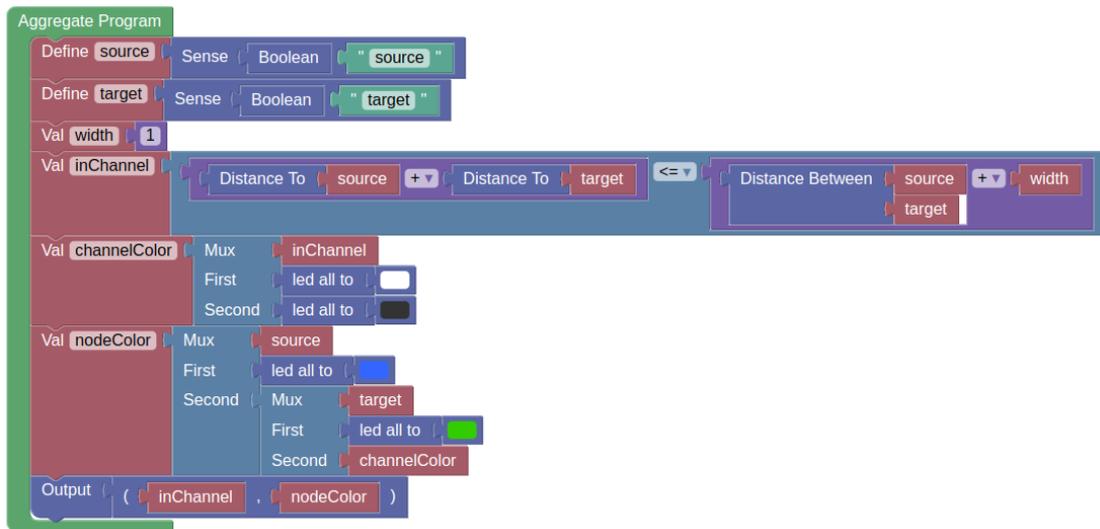


Figura 5.1: L'esempio channel in Blockly2Scafi.

Listato 5.1: Codice ScaFi generato dall'esempio channel in Blockly2Scafi

```

1 //using StandardSensors, BlockG, Actuation
2 def source : Boolean = sense("source")
3 def target : Boolean = sense("target")
4 val width : Integer = 1
5 val inChannel : Boolean = distanceTo(source) + distanceTo(target)
6   <= distanceBetween(source, target) + width
7 val channelColor = mux(inChannel){
8   ledAll to "#ffffff"
9 }{
10  ledAll to "#333333"
11 }
12 val nodeColor = mux(source){
13   ledAll to "#3366ff"
14 }{
15   mux(target){
16     ledAll to "#33cc00"
17   }{
18     channelColor
19   }
20 }
(inChannel, nodeColor)

```

5.2 Verifica dei requisiti

Sono soddisfatti tutti i requisiti funzionali e anche i requisiti non funzionali definiti rispettivamente in 3.1 e 3.2 . In dettaglio:

- R1** Blockly è compatibile con: Chrome, Firefox, Safari, Opera e IE [1].
- R2** Blockly2Scafi offre l'interfaccia delle API in Scala.js (4.4).
- R3** L'applicazione web (fig. 4.3) ha un layout responsive, semplice ed oggettivamente intuitivo.
- R4** La possibilità di realizzare l'esempio Channel in Blockly2Scafi è stata verificata nella sezione 5.1.
 - R4.1** Le operazioni numeriche sono definibili tramite il blocco `number_operation`.
 - R4.2** Le comparazioni numeriche sono definibili tramite il blocco `number_compare`.
 - R4.3** Le variabili possono essere definite tramite i blocchi `val` e `def`.
 - R4.4** I valori dai sensori possono essere ottenuti tramite il blocco `sense`.
Il tipo di valore restituito del sensore deve essere indicato tramite un blocco `type` o `other_type`.
 - R4.5** I blocchi `distance_to` e `distance_between` permettono di chiamare le funzioni di calcolo delle distanze.
 - R4.6** Il costrutto multiplexer è definibile tramite il blocco `mux`.
 - R4.7** Il colore dei nodi è modificabile restituendo in output il blocco `led_all_to` indicandone il colore tramite `color`.
- R5** Il generatore produce codice ben formattato e ben indentato come mostrato nel listato 5.1, le parentesi vengono utilizzate solo se necessario, come descritto nella sezione 4.3.2.
- RNF1** L'applicazione web genera il codice in tempo reale rispetto alle modifiche dell'editor.
- RNF2** Il codice ScaFi generato importa solo i moduli necessari, come descritto nella sezione 4.3.1.

Capitolo 6

Conclusioni

In conclusione si può dire che Blockly2Scafi semplifica l'approccio alla programmazione ScaFi tramite l'uso di elementi grafici che, connessi tra loro, compongono la definizione di un programma aggregato. La futura integrazione in ScaFi-Web è resa possibile grazie all'interfaccia API offerta in Scala.js. Nonostante Blockly2Scafi ancora non supporta molte delle funzionalità offerte dalla controparte testuale, si è rivelato un buon punto di partenza verso lo sviluppo di un linguaggio visuale completo per ScaFi.

Blockly si è dimostrato essere un valido strumento grazie all'alto livello di personalizzazione offerto. La community di Blockly è attiva ed il progetto è in continuo sviluppo. Ad oggi le funzionalità offerte sono più che sufficienti ed esistono, inoltre, decine di plugin first party.

6.1 Possibili sviluppi

Durante lo sviluppo del progetto sono emerse diverse idee e necessità su possibili sviluppi futuri:

- **Possibilità di definire le funzioni e le lambda:** ScaFi fa largo uso della programmazione funzionale, da cui nasce la necessità del VPL di poter definire le funzioni e di poterle utilizzare sia come valore che come procedura.
- **Linguaggio ibrido:** Sarebbe interessante approfondire i meccanismi del linguaggio ibrido, integrando nella programmazione visuale anche una parte di programmazione testuale, ad esempio aggiungendo la possibilità di scrivere codice testuale nei blocchi di definizione delle funzioni.
- **Aumentare le funzionalità di ScaFi offerte dal VPL:** Il VPL ad oggi offre solo un piccolo sottoinsieme delle funzionalità presenti in ScaFi, sarà

necessario aggiungere nuovi blocchi e moduli per permettere la realizzazione di altri esempi d'uso di ScaFi.

- **Importazione automatica dei blocchi derivanti dalle funzioni ScaFi:** un'idea interessante emersa durante lo sviluppo è quella di poter “importare” un set di funzioni di ScaFi e utilizzarle come blocchi. L'importazione sarebbe possibile in quanto i blocchi di chiamata ad una funzione condividono la stessa struttura e si potrebbe generalizzare la generazione del codice. Per definire dinamicamente questi blocchi le informazioni necessarie sarebbero: il nome della funzione; l'elenco dei possibili tipi di valore di ritorno; l'elenco dei parametri; l'elenco dei moduli che è necessario importare; infine, il nome della categoria della toolbox nel quale visualizzare il blocco.
- **Aggiunta di un sistema di warning ed errori** Nonostante il VPL riduca gli errori sintattici, ci sono ancora delle possibilità di effettuarli, ad esempio lasciando vuote delle connessioni che sarebbero obbligatorie oppure definendo due volte una variabile con lo stesso nome si potrebbero generare dei warning ed errori che potrebbero essere visualizzati graficamente con delle icone.
- **Testare l'intuitività del linguaggio visivo** L'aspetto chiave di un linguaggio di programmazione visuale è la sua intuitività. Per riconoscere eventuali barriere di apprendimento, sarebbe interessante far provare il linguaggio a persone che hanno una conoscenza dei principi della programmazione aggregata ma che non conoscano la sintassi del linguaggio ScaFi.

Bibliografia

- [1] Blockly. <https://developers.google.com/blockly>. 2022-02-23.
- [2] Blockly repository. <https://github.com/google/blockly>. 2022-02-23.
- [3] Blockly samples repository. <https://github.com/google/blockly-samples>. 2022-02-23.
- [4] Bluej. <https://bluej.org>. 2022-02-24.
- [5] Kodu games lab. <https://www.kodugamelab.com/>. 2022-02-21.
- [6] Scafi. <https://scafi.github.io/>. 2022-02-18.
- [7] Scafi web. <https://scafi.github.io/web/>. 2022-02-18.
- [8] Scratch. <https://scratch.mit.edu/>. 2022-02-22.
- [9] Snap! <https://snap.berkeley.edu/>. 2022-03-03.
- [10] Gianluca Aguzzi, Roberto Casadei, Niccolò Maltoni, Danilo Pianini, and Mirko Viroli. Scafi-web: A web-based application for field-based coordination programming. In Ferruccio Damiani and Ornella Dardha, editors, *Coordination Models and Languages*, pages 285–299, Cham, 2021. Springer International Publishing.
- [11] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *IEEE Computer*, 48(9):22–30, 2015.
- [12] Alan F. Blackwell, Kirsten N. Whitley, Judith Good, and Marian Petre. Cognitive factors in programming with diagrams. *Artificial Intelligence Review*, 15(1):95–114, Mar 2001.
- [13] Roberto Casadei. *Aggregate Programming in Scala: a Core Library and Actor-Based Platform for Distributed Computational Fields*. PhD thesis.

- [14] Mark Dorling and Dave White. Scratch: A way to logo and python. SIGCSE '15, page 191–196, New York, NY, USA, 2015. Association for Computing Machinery.
- [15] Daniela Giordano and Francesco Maiorana. Use of cutting edge educational tools for an initial programming course. In *2014 IEEE Global Engineering Education Conference (EDUCON)*, pages 556–563, 2014.
- [16] Amy J. Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 199–206, 2004.
- [17] Mohammad Amin Kuhail, Shahbano Farooq, Rawad Hammad, and Mohammed Bahja. Characterizing visual programming approaches for end-user developers: A systematic review. *IEEE Access*, 9:14181–14202, 2021.
- [18] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Trans. Comput. Educ.*, 10(4), nov 2010.
- [19] Mark Noone and Aidan Mooney. Visual and textual programming languages: a systematic review of the literature. *Journal of Computers in Education*, 5(2):149–174, March 2018.
- [20] Erik Pasternak. Scratch 3.0's new programming blocks, built on blockly. <https://developers.googleblog.com/2019/01/scratch-30s-new-programming-blocks.html>, 01 2019.
- [21] Erik Pasternak, Rachel Fenichel, and Andrew N. Marshall. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B B)*, pages 21–24, 2017.
- [22] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *Journal of Logical and Algebraic Methods in Programming*, 109:100486, 2019.
- [23] David Weintrop and Uri Wilensky. To block or not to block, that is the question: Students' perceptions of blocks-based programming. IDC '15, page 199–208, New York, NY, USA, 2015. Association for Computing Machinery.

Ringraziamenti

In questa tesi si è parlato abbondantemente della programmazione visuale e dei vantaggi che offre agli studenti se usata come introduzione al mondo della programmazione, io sono stato uno di loro. Il mio primo approccio alla programmazione è stato proprio con Scratch in prima superiore; senza nessun tipo di esperienza ero riuscito a creare un gioco a livelli dove il personaggio era un cubo che, seguendo il puntatore del mouse, doveva superare un labirinto e schivare ostacoli fino al raggiungimento del traguardo entro lo scadere del tempo. Oggi rigiocando a quel gioco, mi sono reso conto che da lì nacque la mia passione per l'informatica che, passo dopo passo, mi ha portato fino a questo importante traguardo.

Vorrei quindi ringraziare il Prof. Viroli, il Prof. Casadei e il Dott. Aguzzi per avermi proposto l'idea di questa tesi e per avermi guidato nelle scelte progettuali con grande professionalità.

Ringrazio la mia compagna Alma per essere sempre stata al mio fianco da quando ci siamo conosciuti nelle aule dell' Università e per l'infinita pazienza e sostegno che mi ha dato ogni giorno durante la realizzazione di questa tesi.

Ringrazio i miei amici Luca e Tommaso per esserci sempre stati e per tutti i momenti che abbiamo passato insieme.

Ringrazio la società ithings e tutte le persone che ne fanno parte, con le quali lavoro dal primo anno di questo percorso.

Infine, ringrazio la mia famiglia che, nonostante tutto, mi ha reso quello che sono.