POLITECNICO DI TORINO

Corso di Laurea Magistrale in
Ingegneria Informatica

Tesi di Laurea Magistrale

Interfacce per End-User Debugging nel contesto Internet of Things

Relatori:
Fulvio Corno
Luigi De Russis
Alberto Monge Roffarello

Candidata:
Alessia Carosella

Anno accademico 2017-2018

Indice

Capitolo 1

Introduzione

Negli ultimi anni il mondo dell'Internet of Things (IoT) si sta sviluppando sempre più rapidamente, contaminando ogni aspetto della vita sia da un punto di vista personale che industriale.

Possiamo definire "Ecosistema IoT" un sistema formato da dispositivi intelligenti o applicazioni web che interagendo tra di loro riescono a fornire dei servizi agli utenti.

La diffusione di questi dispositivi intelligenti è in costante aumento per due motivi:

1. la costante diminuzione del costo del singolo sensore/dispositivo che li rende di fatto accessibili a gran parte della popolazione e non solo per una cerchia ristretta.

2. il miglioramento delle infrastrutture di comunicazione sia fisse che mobili che rende possibile gestire più dispositivi da remoto in tempo reale.

L'applicazione più comune di sistemi IoT è stata, nel recente passato, nell'ambito della domotica. In questo contesto un sistema IoT era costituito da una serie di dispositivi che un fornitore installava per conto del compratore. Tutti questi dispositivi, dopo essere stati installati, erano controllati da alcune regole suggerite dall'utente e programmate da un tecnico specializzato. La composizione delle regole risultava essere un operazione molto complessa e questo rendeva particolarmente difficile, per un utente finale, modificare il comportamento del sistema senza l'aiuto di un programmatore esperto.

Questo tipo di approccio aveva dei lati sia positivi che negativi.

Da un lato, infatti, permetteva all'utente di evitare di occuparsi di questioni "tecniche" e quindi potenzialmente complesse e pericolose, come l'installazione e la programmazione dei dispositivi. Dall'altro, portava l'utente ad avere la sensazione di non avere un totale controllo sul sistema acquistato. A questo si aggiungeva un

problema di privacy perché all'utente è chiesto di condividere, con il tecnico, dei dettagli sul comportamento desiderato del sistema.

Prendiamo l' esempio di una casa domotica: l'utente per programmare il sistema in modo corretto era costretto a condividere, con un tecnico, dei dettagli sulla propria routine per far sì che quest'ultimo potesse creare delle regole che avessero il comportamento desiderato.

Negli ultimi anni il concetto di come costruire un sistema IoT è cambiato spostando gran parte del lavoro "tecnico" sull'utente finale.

Molte più aziende stanno seguendo il paradigma del Do-it-Yourself (DIY)[24] che vede l'acquirente del servizio come "tecnico aggiunto". È infatti compito del compratore quello di installare e programmare il proprio dispositivo senza aver nessun tipo di ausilio tecnico. Questo paradigma ha ovviamente sia dei risvolti negativi che positivi. Da un lato l'utente finale percepirà come più suo e controllabile il sistema creato e parallelamente l'azienda potrà risparmiare su tutti i costi derivanti dal lavoro degli specialisti. Dall'altro lato però l'utente finale dovrà creare in prima persona le proprie regole e occuparsi della gestione dei problemi che possono generarsi.

Per far sì che anche utenti non esperti potessero completare questo lavoro sono stati creati dei tool specifici per l' End-User Development (EUD).

Questi tools utilizzano dei linguaggi che riescono a rendere molto semplice ed intuitiva la creazione di codice per controllare il sistema in oggetto.

Nell'ultimo periodo, oltre all'applicazione in campo domotico, tool per l'End-User Development sono utilizzati anche per programmare servizi web come social network e servizi di messaggistica. Di conseguenza l'obiettivo di un utente è divenuto quello di programmare congiuntamente dispositivi fisici e servizi virtuali secondo i propri bisogni. Solitamente la gestione di un sistema IoT è fatta mediante la definizione di regole. Queste regole, sono composte secondo il paradigma del Trigger-Action Programmig (paradigma TA) dove ad un evento (trigger) è associata un'azione (action). Ogni trigger o action è offerto da un servizio attraverso un canale. Ad un servizio virtuale corrisponderà ovviamente un canale virtuale e ad un servizio fisico corrisponderà un canale fisico. Ogni qualvolta l'evento si scatenerà, il sistema eseguirà l'azione specificata.

Una regola potrebbe essere ad esempio: *SE il mio smatphone (Android Location) rileva che sono a casa ALLORA accendi le luci (Philiphs Hue) del patio.*

In Tabella 1.1 è possibile trovare la divisione di questa regola in trigger ed action con i relativi servizi e canali di utilizzo.

Nel proseguo della tesi si userà questa notazione per descrivere tutte le regole TA.

	Condizione-Azione	**Servizio**	**Canale**
Trigger	rileva sono a casa	Android Location	mio smartphone
Action	accendi	Philips Hue	luce sul patio

Tabella 1.1: Descrizione regola a basso livello

I tool di EUD permettono ad utenti finali di comporre le regole TA desiderate attraverso l'uso di interfacce grafiche o linguaggi visuali rendendo il compito di gestione dei dispositivi accessibile a tutti, compresi gli utenti senza alcuna conoscenza informatica.

Creare delle regole trigger-action può apparire molto facile ma allo stesso tempo costituisce un operazione molto delicata. Questo perché l'interazione tra le varie regole composte può generare dei problemi che possono provocare dei comportamenti anomali e del falle di sicurezza in sistemi IoT. Per questo motivo le fasi di debug e correzione degli errori risultano essere fondamentali.

Nonostante siano stati fatti molti passi in avanti per quanto riguarda i tool di EUD, resta ancora carente l'assistenza che i questi riescono a fornire durante le fasi di debug.

Prendiamo ad esempio il caso in cui una persona componga queste due regole:

1. *SE il mio smartphone (Android Location) rileva che sono a casa ALLORA accendi la luce (Philips Hue) in cucina*

2. *SE il mio smartphone (Android Location) rileva che sono a casa ALLORA spegni la luce (Philips Hue) in cucina*

In questo esempio le regole composte sono sintatticamente corrette ma la loro interazione scatena un inconsistenza. Questo perché nel caso in cui l'utente dovesse entrare in cucina la luce sarebbe, quasi simultaneamente, accesa e spenta.

L'obiettivo di questa tesi è creare un tool di EUD che possa supportare gli utenti finali ad evitare errori nella composizione di regole, permettendo anche agli utenti meno esperti di eseguire il processo di debug delle proprie regole.

Le domande a cui cercheremo di rispondere sono:

- **QT1)** Quali sono gli errori commessi più frequentemente da un utente non esperto?

- **QT2)** Qual è il processo cognitivo che un utente segue per cercare di risolvere un problema?

- **QT3)** Quale linguaggio visuale risulta essere più versatile ed intuitivo per la composizione e il debug di regole Trigger-Action?

Per fare ciò sarà creato un tool che tramite l'utilizzo di linguaggi visuali possa facilitare la creazione e il debug di regole trigger-action.

Questa tesi si divide in quattro fasi:

1. La prima fase consisterà in una ricerca in letteratura volta a definire quali siano le problematiche principali che gli utenti finali si ritrovano ad affrontare durante l'uso di tool per EUD e quali sono i linguaggi visuali che meglio possano aiutare l'utente nel processo di debug.

2. La seconda fase della tesi verterà sulla progettazione di più soluzioni differenziate tra loro dall'uso di linguaggi visuali diversi. Questa fase sarà condotta tramite la creazione di mockup e alla fine sarà scelta la soluzione migliore.

3. La terza fase consisterà nello sviluppare ed implementare la soluzione scelta in fase due.

4. Nella quarta fase il prototipo creato verrà valutato in uno studio con utenti finali. L'analisi dei risultati di questo studio aiuterà a capire se l'interfaccia creata possa essere d'aiuto o meno per l'utente finale nel capire i problemi che si possono creare e risolverli in maniera adeguata.

Capitolo 2

Lavori Correlati

L'obiettivo di questa tesi consiste nel trovare delle soluzioni che migliorino l'efficacia del processo di debug eseguito da un utente, attraverso la creazione di un tool per l'End-User Development che operi in un contesto IoT. Questo tool dovrà essere in grado di fornire all'utente finale le informazioni di cui necessita durante il processo di debug e di rappresentarle nel modo più intuitivo e chiaro possibile.

Per raggiungere questo obiettivo occorre rispondere a due domande:

- Quali sono le informazioni che un utente reputa essere utili in fase di debug?
- Qual è il linguaggio visuale più adatto per rappresentare queste informazioni?

Partendo da questi aspetti, verrà effettuato uno studio in letteratura per rispondere ai seguenti punti:

1. Quali sono i principali linguaggi visuali utilizzati in ambienti di EUD e in che modo un utente interagisce con essi.
2. Quali sono le maggiori problematiche legate al processo di debug e quali sono le metodologie migliori per supportare un utente durante questa attività.
3. Quali sono le funzionalità, attualmente presenti in tool dedicati all'EUD, che si focalizzano sul processo di Debug.

2.1 Definizione di Linguaggio Visuale

Durante la programmazione di un sistema, un utente non esperto si trova a fronteggiare degli ostacoli come sintassi e semantica. L'uso di linguaggi visuali permette di rimuovere queste barriere grazie alla mediazione, che avviene in modo automatico, tra il modello concettuale e quello computazionale [21].

La definizione di linguaggio visuale non è unica [11] e può avere una doppia valenza. Per alcuni è inteso come un linguaggio che processa informazioni visuali, cioè che gestisce oggetti con una rappresentazione intrinseca. Ad esempio un programma per la gestione e riconoscimento di foto può essere considerato un linguaggio visuale perché processa informazioni, ovvero delle foto, che hanno una rappresentazione visuale intrinseca. In altri casi, invece, i linguaggi visuali sono utilizzati per rendere più intuitivi i linguaggi di programmazione. In questo caso, oggetti come liste o variabili sono rappresentati visualmente secondo il valore logico che questi assumono all'interno dell'ambiente di programmazione.

Questa seconda definizione è quella usata dai tool per l'End-User Development e in questo capitolo definiremo le possibili rappresentazioni che questi linguaggi possono assumere. Saranno quindi analizzati quali sono i principali linguaggi visuali e quali le loro caratteristiche più importanti.

Le rappresentazioni visuali possono essere suddivise in 3 gruppi:

1. Form-Filling
2. Dataflow programming
3. Block-programming

Form-filling La rappresentazione mediante form-filling risulta essere la più usata in campo di End-User Development. L'utente aggiunge delle funzionalità al suo programma selezionando degli elementi attraverso l'uso di menu (Figura 2.1, Figura 2.3) o azioni di tipo drag&drop (Figura 2.2).

Le interfacce basate sul form-filling sono strutturalmente predefinite, hanno un punto di partenza fisso e guidano l'utente obbligandolo ad seguire una procedura predefinita passo dopo passo fino al completamento del task. Il compito dell'utente si riduce alla compilazione di campi, che di volta in volta gli vengono presentati.

Prendiamo ad esempio il caso in cui un utente voglia comporre la seguente regola:
SE il mio smatphone (Android Location) rileva che sono a casa ALLORA accendi le luci (Philiphs Hue) del patio.

Il sistema chiederà all'utente passo dopo passo di selezionare un *trigger* (con le relative specifiche) e di scegliere un *action* (con le relative specifiche).

I passi eseguiti saranno quindi questi:

1. Richiesta di selezione del servizio che offre il *trigger*

   L'utente seleziona *Android Location*

2. Richiesta di selezione del trigger

   L'utente seleziona *Entro a casa*

3. Richiesta di selezione del servizio che offre *action*

   L'utente seleziona *Philiphs Hue*

4. Richiesta di selezione del *action*

   L'utente seleziona *Accendi luce sul patio*

Il meccanismo di scelta dell'utente può essere basato su un'interfaccia visuale [1] , testuale [4], o una combinazione delle due.

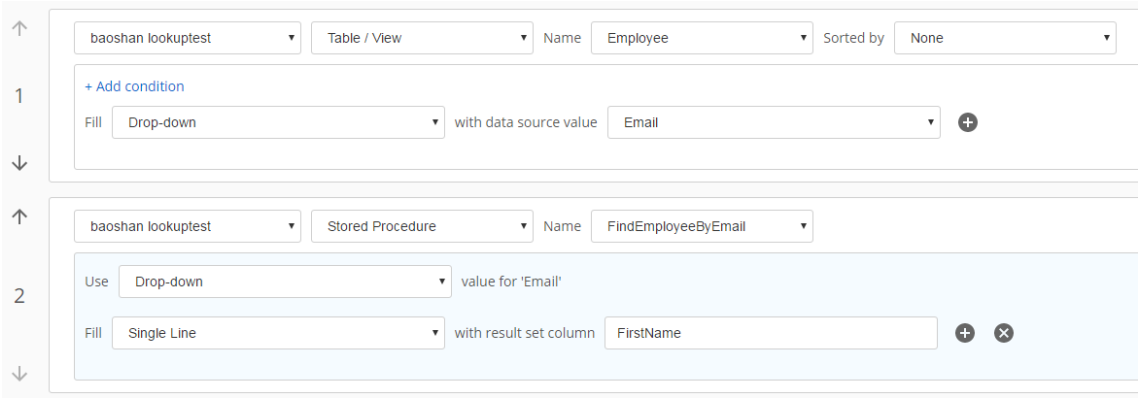


Figura 2.1: Un esempio di interfaccia form-filling con meccanismo di scelta *dropdown*.

Un esempio di form-filling può essere considerato quello offerto da IFTTT (Figura 2.3). IFTTT [1] è un'applicazione web gratuita che permette la creazione di semplici regole per la gestione di servizi virtuali come Facebook o Instagram. In una soluzione di tipo form-filling l'utente usando il click del mouse seleziona di volta in volta un elemento tra quelli proposti e passo dopo passo arriva a formare la regola finale in tutte le sue specifiche.

Dagli studi compiuti [20, 19] si evince che il form-filling risulta essere un rappresentazione molto intuitiva e semplice che permette ad un utente di raggiungere facilmente un obiettivo.

Nel caso di task semplici e quindi con pochi elementi da gestire, la natura restrittiva del linguaggio viene percepita da un utente finale in modo positivo, perché questo fornisce ad un utente delle sicurezze sulla procedura da eseguire.

Nel caso in cui i task risultino essere più complessi ed articolati, richiedendo l'uso di più elementi, la natura restrittiva del linguaggio non risulta sempre gradita dall'utente e le difficoltà incontrate nel raggiungimento dell'obiettivo finale aumentano.

Figura 2.2: Un esempio di interfaccia form-filling con meccanismo di scelta *drag&drop*.

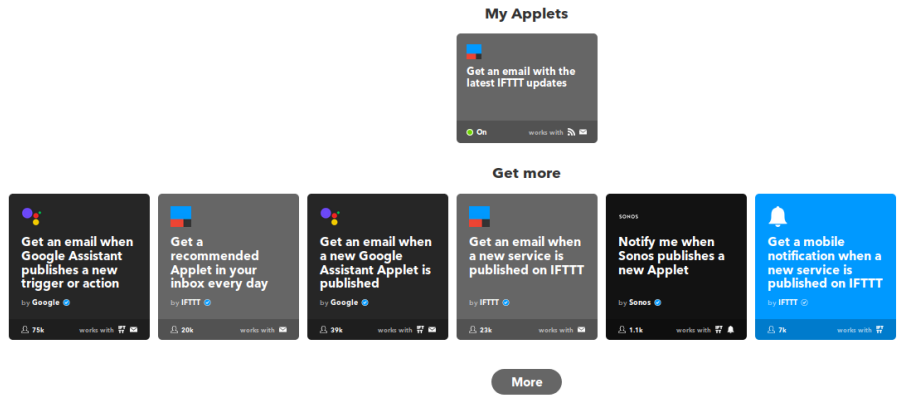


Figura 2.3: interfaccia del sistema IFTTT form-filling

In questi casi, secondo quanto riscontrato da alcuni test [20], gli utenti tendono, infatti, a non ricordare correttamente quali scelte sono state compiute in passato.

Dataflow programming Il Dataflow programming risulta essere uno dei linguaggi visuali utilizzati più di rado. Possiamo trovarne degli esempi in interfacce che si occupano di creazione di diagrammi o in tool che supportano editor d'esecuzione interattivi. Questo linguaggio visuale (Figura 2.4) è composto da due ambienti:

1. Uno spazio aperto da utilizzare come area di lavoro

2. Uno spazio utilizzato per la visualizzazione dell'insieme degli elementi disponibili divisi per categorie (repository).

L'utente usando operazioni drag&drop può spostare gli elementi dal repository all'area di lavoro e successivamente collegare più blocchi tra loro attraverso l'uso di linee (wires) che ne rappresentano le connessioni logiche.

Il dataflow programming è un linguaggio strutturalmente aperto, con un punto di inizio non predefinito e senza alcun ordine di esecuzione. L'utente sarà quindi incaricato di gestire tutto il processo di programmazione decidendo in prima persona quali siano i passi da eseguire.

Prendiamo ad esempio il caso in cui il nostro utente voglia comporre la seguente regola:

SE il mio smatphone (Android) rileva che sono a casa ALLORA accendi le luci (Philiphs Hue) del patio.

L'utente in questo caso non sarà guidato dall'interfaccia nella selezione dei blocchi o nella creazione delle relazioni ma potrà decidere in modo autonomo quale strategia utilizzare per arrivare al raggiungimento del proprio obiettivo.

Gli step da eseguire, non necessariamente in ordine, saranno:

1. L'utente seleziona il servizio che offre il trigger tra la lista di servizi disponibili.
2. L'utente seleziona un *trigger* dal repository e lo trascina nell'area di lavoro.
3. L'utente seleziona il servizio che offre l'action tra la lista di servizi disponibili.
4. L'utente seleziona un *action* dal repository e lo trascina nell'area di lavoro.
5. L'utente crea una connessione logica creando una freccia che parta dal *trigger* per arrivare *action*.

Dai risultati di alcuni test con utenti [20, 19] si evince come l'utente, utilizzando la programmazione dataflow, avverta maggiori difficoltà nel raggiungimento dell'obiettivo. Risulta infatti difficile capire quali siano i blocchi da utilizzare e quale sia la giusta connessione logica con cui collegarli. Se da un punto di vista la mancanza di rigidezza d'esecuzione provoca smarrimento iniziale tuttavia, dall'altro nel caso in cui i task diventino più complessi si può notare come una visualizzazione mediante dataflow risulti essere molto più chiara rispetto a quella fornita dal form-filling perché le relazioni tra tutti i blocchi risultano essere più evidenti.

Anche se percepito come generalmente più complesso, alcuni partecipanti agli studi hanno osservato come questo tipo di linguaggio fosse molto creativo, interessante e, di come gran parte della difficoltà iniziale risiedesse nella mancanza di conoscenza del tool. Molti utenti hanno in fine notato come il dataflow

programming facilitasse una visione d'insieme permettendo di vedere più regole contemporaneamente.

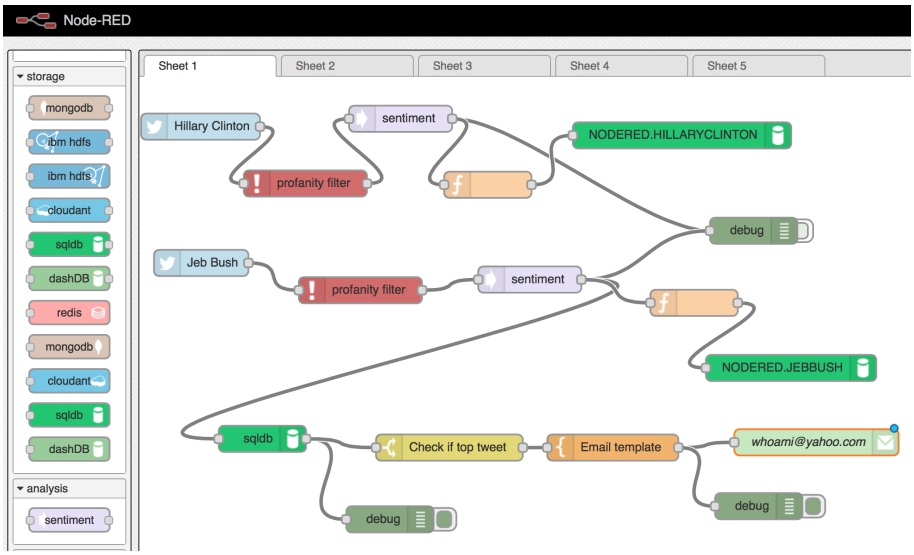


Figura 2.4: Esempio di dataflow programming

Block-programming Il block-programming è un linguaggio visuale largamente utilizzato nel mondo dell'End-User Development.

Può avere diverse rappresentazioni visuali dalla più comune a blocchi a quella che utilizza una jigsaw metaphor. Altre visualizzazioni, meno usate, possono essere quelle del card sorting oppure quella di un linguaggio pseudo-naturale. Indipendentemente dal tipo di rappresentazione visuale utilizzato, l'interazione avviene tramite un meccanismo basato sul drag&drop. La jigsaw metaphor consiste nell'uso di blocchi aventi la forma di pezzi di un puzzle, mentre le connessioni logiche sono create in modo automatico quando più pezzi sono aggancianti tra di loro.

Il block-programming è, come il dataflow programming, un linguaggio strutturalmente aperto, con un punto di inizio non predefinito e senza alcun ordine di esecuzione.

L'interfaccia è composta, come nel caso del dataflow programming, da due aree, un'area di lavoro, e il repository dei blocchi disponibili.

Gli oggetti presenti nelle repository possono essere trascinati nell'area di lavoro e quando due o più oggetti sono agganciati tra di loro vengono create le connessioni logiche.

Anche in questo caso l'utente non è guidato nella completamento del task step-by-step ma è libero di comporre il programma nel l'ordine da esso preferito.

Grazie all'utilizzo della jigsaw metaphor l'utente visualizzerà i trigger e le action come blocchi diversi e complementari, non avrà quindi alcuna difficoltà nel capire la connessione logica esistente tra i due, rimuovendo di fatto le limitazioni trovate nel dataflow programming.

La rappresentazione mediante puzzle stimola gli utenti a combinare funzioni già esistenti per raggiungere il loro obiettivo. Questo tipo di approccio permette di porsi ad un livello di astrazione molto più alto rispetto a quanto riescono a fare gli altri linguaggi visuali poiché il linguaggio non si limita a usare una rappresentazione visuale di concetti logici, ma mette a disposizione dell'utente dei blocchi (pezzi di puzzle) già pronti per essere combinati insieme. Inoltre la loro forma rende molto chiaro per l'utente finale capire come combinare i vari pezzi tra loro.

Grazie a questo approccio risulta molto più facile comporre degli algoritmi complessi, per questo motivo la jigsaw metaphor è il linguaggio visuale più usato da tool che attraverso rappresentazioni visuali permettono ad utenti finali di creare programmi molto complessi come ad esempio videogame [3, 2] o story-telling [15].

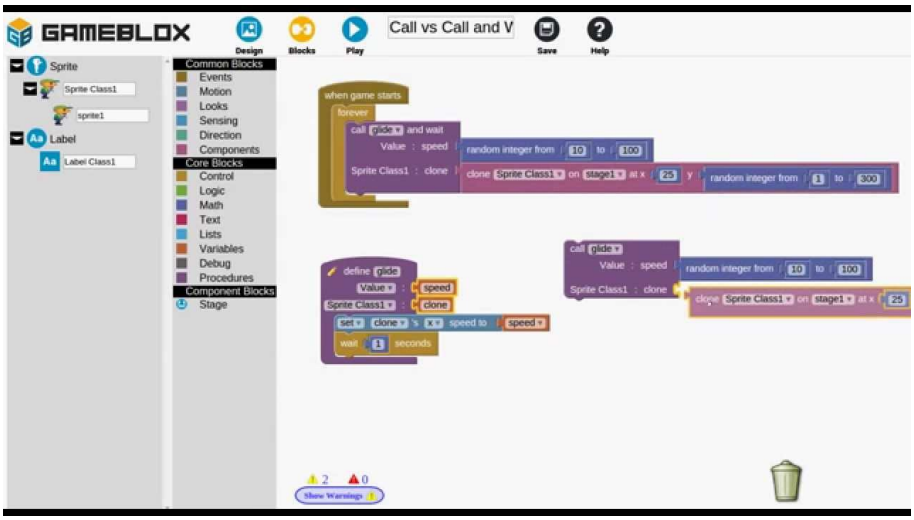


Figura 2.5: Esempio di block programming -GameBlox

Considerazioni finali Concludendo questa prima fase di ricerca i risultati ottenuti ci dicono che:

1. Il form-filling è il linguaggio più chiaro ed intuitivo ma poco adattabile a situazioni con un maggior numero di elementi.

2. Il dataflow programming risulta essere poco intuitivo in fase di composizione ma molto utile in fase di revisione grazie alla chiarezza della sua visione di insieme.

3. Il block-programming, rappresentato sotto forma di jigsaw metaphor, risulta essere molto chiaro e facilmente adattabile nel caso in cui il numero di elementi non sia limitato, fornendo quindi una user-experience più stimolante.

2.2 Debug per utenti finali

Il debug è per definizione l'attività di individuazione e correzione di uno o più bug, ovvero errori logici o sintattici, presenti in un software. Questo, anche quando effettuato da utenti esperti, risulta essere una delle attività più complesse e delicate fatte in fase di programmazione.

L'attività di debug è svolta utilizzando tecniche diverse a seconda del grado di esperienza del programmatore che la esegue.

Un programmatore, esperto o inesperto, si focalizzerà inizialmente sull'ottenere una conoscenza ad alto livello del sistema e delle sue interazioni prima di iniziare il processo di debug. Al contrario gli utenti finali inizieranno direttamente dalla fase di correzione dei bug senza possedere una visione complessiva del sistema chiara e completa.

Questa differenza di approccio è strettamente legata al tempo che ogni persona è disposta ad impiegare per il raggiungimento di un obiettivo finale. L'obiettivo di un programmatore inesperto sarà quello di imparare e quindi il tempo impiegato per la fase iniziale di conoscenza della struttura del sistema non è percepito in maniera negativa. Al contrario, l'obiettivo di un utente finale sarà quello di completare il task nel minor tempo possibile e di conseguenza il tempo utilizzato in attività senza un riscontro immediato è percepito in maniera negativa [22].

Un'ulteriore difficoltà è data dalla possibilità che un utente finale possa non accorgersi della presenza di un bug o non sapere qual è il modo corretto per risolverlo.

Per diminuire le difficoltà in fase di debug nell'End-User programming occorre creare dei tools che assistano l'utente prendendosi carico di parte del processo stesso di debug.

La creazione di un tool di questo tipo rappresenta l'obiettivo finale di questa tesi e in questa sezione ci occuperemo di ricercare in letteratura:

- Gli errori più frequenti nel quale un utente può imbattersi durante le operazioni di debug.

- Il processo cognitivo di un utente non esperto durante la fase di debug.

2.2.1 Gli errori comuni

Gli errori più frequenti fatti in fase di programmazione, indipendentemente dal grado di esperienza di chi li commette, sono dovuti tipicamente a:

1. Errori di sintassi
2. Ipotesi contestuali errate

   Es. *Pensare che l'unità di misura usata da un sensore sia in gradi Celsius invece di Fahrenheit*
3. Errore nel settaggio di parametri

   Es. *Nella compilazione dei campi per l'invio di una mail inserire il destinatario nello spazio riservato all'oggetto.*
4. Dataflow composti da regole caratterizzare da effetti inconsistenti o errati.

   Es. *Creare una regola la cui interazione con le altre genera loop,inconsistenze o ridondanze.*

Possiamo dividere questi errori in due categorie:

- La prima categoria conterrà tutti gli errori dovuti alla mancanza di dimestichezza con linguaggio o tool utilizzato. (Errori 1 e 3)
- La seconda categoria invece conterrà tutti gli errori che derivano da valutazioni contestuali sbagliate da parte dell'utente (Errori 2 e 4).

Concentrandoci sugli errori appartenenti alla seconda categoria andremo ora a capire in quale modo un utente finale interagisce essi. Per fare ciò è importante andare ad indagare il processo cognitivo che un utente segue per risolvere un problema.

2.2.2 L'errore visto da un utente finale

In letteratura sono stati fatti molti studi per capire qual è l'approccio che un utente finale segue per risolvere un problema. Gli studi sono stati portati avanti su vari campi dell'End-User Development come quelli che si occupano di correzione di fogli di lavoro [14], creazione di siti con tecnica di mash-up [9] e creazione di giochi o story-telling per bambini [15].

Tutti gli studi hanno appurato che un utente finale cerca di risolvere i problemi utilizzando un metodo chiamato *debugging into existence*. Questo metodo si basa su una continua ricerca di bug e una conseguente creazione di codice per la correzione degli stessi [9, 14].

Questo ciclo risulta essere diviso in 3 fasi [10]:

- *Framing*

  Durante questa fase l'utente capirà e definirà il problema creando quindi tutte le ipotesi del modello.

- *Acting*

  Durante questa fase l'utente cercherà di ottenere più informazioni sulla situazione corrente. Cercherà inoltre di capire quali sono le trasformazioni da apportare al modello per migliore la condizione attuale.

- *Reflecting*

  Durante questa fase l'utente cercherà di capire quali sono le conseguenze che la modifica fatta in fase di acting ha portato sul modello.

Queste tre fasi vengono ripetute in modo ciclico passando dal framing all'acting al reflecting. In alcuni casi l'utente può anche decidere di fare quello che è chiamato *major refremnig* che corrisponde ad un cambiamento delle ipotesi contestuali fatte in fase di definizione del problema iniziale.

Lo studio [10] ha analizzato il comportamento di alcuni utenti durante la creazione di un sito utilizzando la tecnica di mashup. In questo caso gli utenti hanno utilizzato la piattaforma Popfly che consente di creare mashup attraverso l'uso di dataflow con l'uso di blocchi. È stato notato che spesso una fase di framing positiva risulta essere fortemente correlata alla quantità di informazioni che l'utente ottiene in seguito a delle interazioni con il tool. Infatti, ogni qual volta, queste interazioni avvenivano con successo l'utente risultava essere in grado di generare delle idee da implementare durante la fase di action. Al contrario quando le interazioni con il tool risultavano essere assenti l'utente incontrava un barriera progettuale, ovvero non riusciva a stabilire quali fossero le azioni necessarie per il raggiungimento del proprio obiettivo.

Da questa analisi si evince quanto sia importante per un tool fornire una guida durante la fase di framing per aiutare l'utente a superare ogni tipo di barriera progettuale.

Un altro approccio che risulta essere molto usato è quello del *tinkering with reflection*. In questo tipo di approccio l'utente crea varie combinazioni con gli oggetti che ha a disposizione e di volta in volta seleziona le combinazioni con un risultato più vicino a quello prefissato.

Questo approccio può essere considerato un *brute force approach* ovvero un tentativo da parte dell'utente di provare tutte le idee che riesce a generare senza filtrarle da un punto di vista empirico. Affinché questa tecnica abbia esiti positivi è importante che l'utente possa testarle frequentemente. Per questo motivo è importante che il "costo del test" sia basso.

Il problema del testing frequente non è rilevato solo in caso di *tinkering* ma in generale questo risulta essere correlato al completamento dei task assegnati.

Dallo studio [10] si evince, inoltre, che nei casi in cui l'utente testava frequentemente la propria soluzione aumentava la percentuale di task completati. Al contrario gli utenti che testavano meno frequentemente andavano spesso a scontrarsi con barriere di comprensibilità del problema("non so perché il problema si comporta in questo modo").

Un'altro fattore importante è dato dalla difficoltà di attuare una politica di "back-track" ovvero di ripristinare uno stato precedente dopo aver apportato alcune modifiche. Per questo motivo è importante che il tool supporti un ripristino della situazione precedente per evitare che gli utenti cerchino di ricrearla da soli utilizzando solo la propria memoria.

Un altro punto importante emerso è quello della self-efficacy ovvero la fiducia che l'utente pone nelle proprie capacità. Gli utenti con bassa self-efficancy risultano essere quelli con una fase di framing meno positiva e che risultano essere riluttanti al provare nuove idee o cercare delle soluzioni alternative. La paura di sbagliare porta questi tipi di utente a cercare di creare delle soluzioni a partire da esempi o suggerimenti ricevuti dal tool senza però capire realmente quale sia il contesto in cui si trova. Questo comportamento è chiamato *inflexibility* e per arginare questo comportamento é meglio evitare di fornire all'utente delle soluzioni comuni per il problema che si trova a risolvere e invece fornire delle informazioni di contesto nel modo più chiaro possibile.

Un altro dato importante da tenere in considerazione è la differenza di approccio tra donne e uomini. Le donne sono caratterizzate da una minore self-efficancy e maggiore inflexibility al contrario degli uomini. Questa predisposizione influenza ovviamente l'approccio al problema che si ha [14]. Le donne sono portate a fare un'analisi più comprensiva del problema cercando di avere una conoscenza di insieme, al contrario gli uomini preferiscono utilizzare un approccio euristico volto a massimizzare le performance, cercando quindi di analizzare solo le informazioni che sono reputate necessarie per la risoluzione del problema.

Queste differenze si propagano anche nella metodologia usata nel debug. Si può notare come le donne preferiscano usare tecniche di *code inspection* in modo da controllare ogni dettaglio implementativo mentre gli uomini siano più propensi all'utilizzo di rappresentazioni visive che riassumano le dipendenze dei vari moduli.

Dall'analisi di tutti gli studi presentati si evince la necessità di progettare un tool che:

- In fase di framing sappia fornire delle informazioni utili sul contesto e suggerire delle possibili idee all'utente

- In fase di acting e reflecting invogli l'utente a testare frequentemente la propria soluzione e fornisca dei meccanismi che permettano modifiche temporanee e ripristini

2.3 Debug con linguaggi visuali

Recentemente molti tool per End-User Debugging hanno iniziato a focalizzare la loro attenzione sul processo di debug, ideando e implementando funzionalità diverse che possano fornire assistenza ad un utente durante questa fase.

Alcuni esempi di queste soluzioni, che andremo ad analizzare in questa sezione, sono il WYSIWYT con localizzazione degli errori per fogli di lavoro [16, 6, 8], la WhyLine per sistemi basati su eventi come Alice 2.0 [17, 15, 18] e il progetto Idea Garden [8, 7].

Il compito di questi sistemi è quello di fornire ad un utente tutto l'aiuto necessario per superare delle barriere create a causa dell'*information gap*. Per information gap si intendono tutte le informazioni che un utente non ha in fase di programmazione e che lo portano a fare degli errori e creare quindi dei bug. Le informazioni possono essere di natura concettuale (es. Quale strategia usare per debuggare) o contestuale (es. Non capisco perché non funziona)

Tutti i tool di EUD implementano queste funzionalità usando diversi linguaggi visuali in modo da fornire la migliore esperienza utente possibile.

What You See Is What You Test Una soluzione molto utilizzata da tool che si occupano di correzione di fogli di lavoro è quella del *What You See Is What You Test* (WYSIWYT).

Il WYSIWYT è composto da un insieme di funzionalità che permettono all'utente di marcare in modo incrementale come corretto o errato il valore di un determinata cella. Partendo da questi valori il sistema va a misurare la qualità del processo di testing eseguito dall'utente secondo dei criteri prestabiliti. I risultati prodotti sono poi mostrati all'utente attraverso l'uso di colori, icone e in alcuni casi rappresentazioni più complesse come un dataflow.

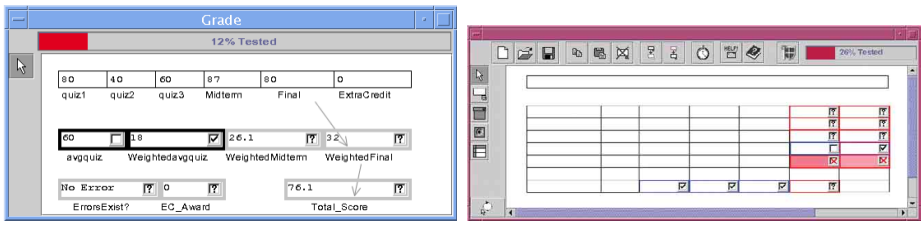
Ad ogni cella viene associato un colore e delle icone che ne segnalano lo stato (Figura: 2.6b).

I possibili stati possono essere quattro: testato, non testato, valore corretto, valore errato.

In aggiunta è possibile visualizzare, con l'ausilio di un dataflow, le relazioni che legano le varie celle del foglio di lavoro (Figura:2.6a).

L'uso del dataflow risulta essere molto utile, perché l'uso di frecce e colori consente di guidare un utente attraverso il processo di debug senza però influenzare in modo eccessivo il loro processo cognitivo naturale.

In alcuni casi il WYSIWYT è stato utilizzando contemporaneamente ad un 'altra funzionalità chiamata *Help Me Test*. L'Help Me Test permette all'utente di testare in modo sistematico il valore di alcune celle utilizzando come input dei valori generati in modo casuale dal sistema. L'unico compito di un utente sarà quindi quello di valutare l'output fornito dal codice utilizzando le funzionalità offerte dal WYSIWYT.



(a) Esempio di dataflow (b) Esempio di foglio da lavoro

Figura 2.6: Esempio di WYSIWYT

WhyLine Sono stati condotti numerosi studi sul tool di Alice 2.0 [15] per capire quali fossero le domande più frequenti che un utente si poneva in fase di debug.

Alice2.0 è un tool progettato per permettere a dei bambini di circa 5-6 anni di sviluppare delle storie animate utilizzando un linguaggio visuale di tipo block-programming. I bambini trascinando dei blocchi e concatenandoli, possono creare delle storie e vedere il risultato ottenuto attraverso una simulazione.

Analizzando il processo di debug, eseguito dagli utenti in questo ambiente, è emerso che le domande più frequenti, fatte durante il processo di debug, fossero "Perché è successo questo?" e "Perché non è successo questo?".

Per rispondere a queste domande è stato ideato un nuovo paradigma di debug chiamato *Interrogative Debugging*. L'obiettivo dell'Interrogative debugging è quello di rispondere a tutte le domande del tipo "Perché è successo l'evento X?" o "Perché non è successo l'evento Y?" creando delle risposte partendo dalle informazioni che il sistema ha sull'ultima esecuzione del programma.

È possibile associare alla domanda del tipo "Perché è successo l'evento X?" l'esecuzione di un azione inaspettata.

Questa condizione si può verificare in due casi:

- Condizione invariante

  In questo caso la condizione che scatena quell'azione risulta essere sempre verificata. Questo può essere dovuto ad esempio da una condizione errata.

- Condizione dovuta all'interazione di altre regole

  In questo caso la concatenazione di più eventi porta all'esecuzione di quell'evento.

È possibile associare alla domanda del tipo "Perché non è successo l'evento X?" la mancata esecuzione di un'azione attesa durante la fase di esecuzione del programma. In questo caso le possibili cause del problema possono essere riconducibili a tre categorie:

- Affermazione errata

  In questo caso l'azione attesa risulta essere stata eseguita dal sistema ma la sua esecuzione è risultata essere invisibile agli occhi dell'utente.

- Condizione invariante

  In questo caso la condizione che scatena l'azione atteso risulta non essere mai verificata. Questo può essere causato ad esempio da una condizione errata.

- Condizione dovuta all'interazione di altre regole

  In questo caso la concatenazione di più eventi ha evitato che l'azione fosse eseguita.

Qualunque sia il tipo di domanda posta le risposte conterranno solo un sottoinsieme di tutte le azioni eseguite dal programma. Saranno infatti mostrate all'utente solo le azioni che risultano essere connesse con quella sulla cui esecuzione l'utente ha chiesto maggiori spiegazioni.

È possibile trovare un applicazione di Interrogative Debugging in Alice2.0 [17]. In Alice 2.0 è stata, infatti, implementata una funzione chiamata Whyline (Workspace that Helps You Link Instructions to Numbers and Events) che permette agli utenti di fare delle domande sulla mancata esecuzione, o meno, di una determinata azione.

In Alice2.0 un utente può formulare una domanda utilizzando un sistema form-filling con selezione tramite menu.

La risposte create dal sistema verranno mostrate all'utente usando due rappresentazioni diverse e complementari:

- Simulazione grafica

  In questa rappresentazione l'utente potrà visualizzare una simulazione delle azioni eseguite durante l'esecuzione del programma tramite l'uso di animazioni.

- Dataflow

  In questa rappresentazione il sistema rappresenterà le azioni che compongono la risposta alla domanda posta dall'utente mediante una rappresentazione dataflow.

  Il dataflow è ordinato su una base temporale e attraverso lo spostamento di un cursore consente all'utente di controllare in tempo reale il cambiamento sia delle variabili presenti all'interno del dataflow che della simulazione.

  Per ciascun elemento rappresentato nel dataflow sarà possibile richiedere più informazioni utilizzando la Whyline nella forma di "Cosa ha causato l'esecuzione di questo elemento". In questo caso il sistema rileverà la sequenza di azioni che ha portato l'esecuzione di quella richiesta e lo rappresenterà sotto forma di dataflow .

Nello studio[15] si è notato come l'uso della Whyline abbia migliorato le performance riducendo sia il tempo impiegato a completare un task che il suo tasso di successo.

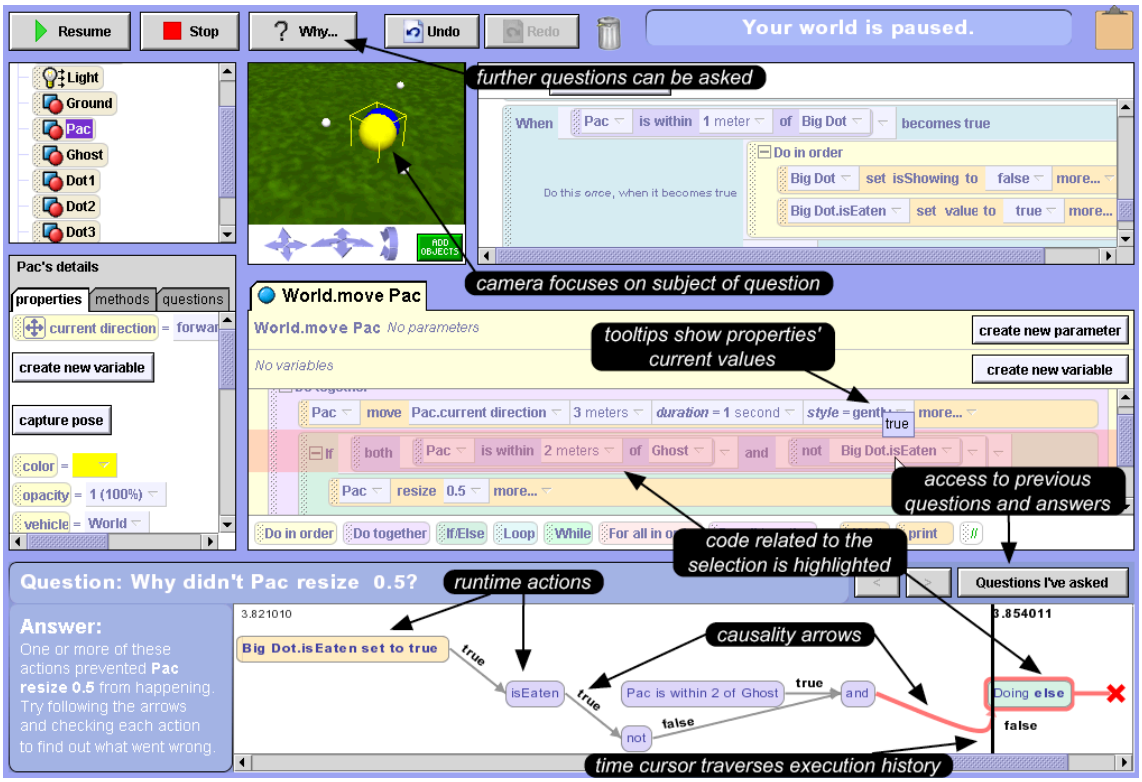


Figura 2.7: Implementazione della WhyLine in Alice2.0

Idea garden L'approccio Idea Garden ha come obiettivo quello di educare gli utenti a affrontare i problemi del debug in modo autonomo grazie all'utilizzo di un supporto di nozioni di problem-solving a richiesta.

Idea Gardern trae origine a partire dalla teoria di "problem-solving" di Simon [23] e dal *"Minimalistic Learning Theory"* [13], il quale stabilisce che affinché un utente percepisca in maniera efficace dei suggerimenti questi devono contenere solo delle informazioni strettamente necessarie.

Idea Garden ha quindi come obiettivo quello di fornire ad un utente delle strategie di debug sotto forma di suggerimenti (Figura 2.8).

Il compito di un sistema che implementa questo approccio è quello di cercare di capire in base allo stato del sistema e le ultime operazioni dell'utente quale può essere una strategia di debug corretta. La rappresentazione del suggerimento è solitamente sotto forma testuale.

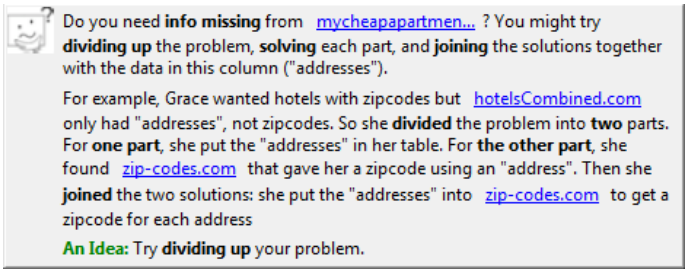


Figura 2.8: Esempio di tooltip generato da Idea Gardern.

Un esempio possibile di funzionamento può essere dato dalla situazione in cui:

- l'utente dopo aver creato un programma molto complesso, attraverso l'uso di molte funzioni ed elementi, blocca il suo operato interrompendo l'interazione con il tool per 15 minuti;
- il tool suggerisce all'utente un approccio di tipo "divide et impera" (Figura 2.8)

L'obiettivo finale di questo approccio non è di fornire un aiuto ad un utente durante le operazioni debug, ma è quello di fornire ad un utente una metodologia corretta per la fase di debug che possa essere utilizzata per risolvere i problemi in maniera autonoma.

Affinché questi suggerimenti siano recepiti in maniera corretta dall'utente è importante capire sia cosa mostrare che il linguaggio visuale da utilizzare.

Un fattore importante risulta essere la lunghezza giusta del messaggio da mostrare all'utente. Nel caso in cui questa sia troppo corta il suggerimento potrebbe risultare essere poco chiaro, nel caso in cui il messaggio sia troppo lungo potrebbe

portare l'utente ad ignorarlo perché in questo caso il costo di lettura risulterebbe troppo alto e anche nel caso in cui l'utente decidesse di leggerlo le informazioni da processare potrebbero essere troppe. Infatti, visto quanto suggerito dal modello di "*Attention Investment*" [5], la probabilità che un utente investa del tempo nell'uso di un componente è strettamente legato al costo percepito in termini cognitivi e temporali.

Per quanto concerne le rappresentazioni visuali esistono varie possibilità. È stato notato [7] come l'uso dei pop-up non sia un modo corretto. Al contrario risulta avere un riscontro molto positivo l'uso di una strategia *Surprise-Explain-Reward*.

L'obiettivo di questa strategia è quello di incuriosire l'utente con dei brevi messaggi o l'uso di colori. Sarà poi compito dell'utente, se interessato, a chiedere al sistema maggiori informazioni. Anche in questo caso la rappresentazione visuale da utilizzare per rappresentare le informazioni è solitamente quella testuale.

Capitolo 3

Design e Progettazione

In questo capitolo, tenendo conto di quanto appreso nel corso dello studio in letteratura (capitolo 2), verranno proposte delle soluzioni specifiche per il raggiungimento dell'obiettivo di questa tesi, ovvero la creazione di un tool di End-User Development che riesca ad assistere gli utenti nella composizione e debug di regole in un contesto IoT.

Inizialmente verrà condotta un'analisi preliminare per identificare gli errori più comuni commessi da utenti finali durante il processo di composizione di regole trigger-action. Successivamente, utilizzando le metodologie descritte nella sezione 2.2, verranno definite delle funzionalità per aiutare un utente durante la fase di debug.

Dopo aver completato l'analisi preliminare avrà inizio la fase progettazione di una possibile applicazione web.

La progettazione sarà suddivisa in due fasi:

1. design di mockup;

2. progettazione dell'infrastruttura e del processo di debug che andranno a formare il prototipo finale.

3.1 Analisi preliminare

Focalizzando l'analisi sugli errori che un utente può commettere durante la composizione di una regola, utilizzando il paradigma di programmazione TA, sono state identificate diverse tipologie di errori:

1. errori sintattici che portano alla definizione di una regola sintatticamente errata, ovvero, composta da soli trigger o sole action.

2. Creazione di loop, ovvero, creazione di un set di regole la cui esecuzione risulti essere infinita e senza una fase di stallo. Ad esempio un loop è generato in seguito al salvataggio delle regole:

   - "*Se creo un post (Facebook) allora crea un tweet (Twitter)*"
   - "*Se creo un tweet (Twitter) allora crea un post (Facebook)*"'.

   In questo caso il sistema dal momento dell'attivazione di una delle due regole pubblicherà contenuti su Twitter e Facebook per sempre, senza arrivare mai ad uno stallo.

3. Creazione di inconsistenze, ovvero, creazione di un set di regole la cui esecuzione genera effetti inconsistenti tra loro. Un esempio di inconsistenza è generato in seguito al salvataggio delle regole:

   - "*Se il mio smartphone rileva che sono a casa (Android Location) allora accendi la lampada della cucina (Philiphs Hue)*"
   - "*Se il mio smartphone rileva che sono a casa (Android Location) allora spegni la lampada della cucina (Philiphs Hue)*"

   In questo caso il sistema accenderà e spegnerà la lampada della cucina in modo quasi simultaneo, creando un effetto impredicibile.

4. Creazione di ridondanze, ovvero, creazione di un set di regole la cui esecuzione genera effetti ridondanti tra loro. Un esempio di ridondanza è generato in seguito al salvataggio delle regole:

   - "*Se creo un post con una foto (Facebook) allora crea un tweet (Twitter)*"
   - "*Se creo un post (Facebook) allora crea un tweet (Twitter)*"

   In questo caso la condizione di attivazione delle due regole è la stessa e il sistema, quando questa si verificherà, eseguirà, in modo quasi simultaneo entrambe le regole. Questo comporterà la pubblicazione di due tweet identici sullo stesso account in seguito alla stessa azione.

5. Errori concettuali. In questo caso le regole risulteranno essere sintatticamente giuste e non creeranno problemi di interazione ma l'esecuzione di queste risulterà comunque errata. Questo perché l'utente avrà commesso un errore concettuale in fase di composizione creando una regola che non ha il comportamento desiderato indipendentemente dai fattori esterni che la possono influenzare.

Per gestire questi errori è possibile definire delle funzionalità basate su alcune delle metodologie descritte nel capitolo 2.

Errori sintattici Per evitare che un utente commetta errori sintattici durante la fase di composizione della regola (errore 1), possono essere utilizzati dei linguaggi visuali come form-filling, dataflow e block-programming.

Costo di esecuzione Come tutti i tool che si occupano di creazione di regole per gestione di dispositivi e servizi web, anche il nostro sistema, dovrà fare particolare attenzione al "costo di esecuzione". Creare un sistema con un "costo di esecuzione" basso risulta essere molto importante per limitare l'impatto di errori concettuali che costituiscono l'errore 5.

Come nel caso della DiY Home, infatti, il "costo di esecuzione" può essere potenzialmente molto alto nel caso in cui le regole composte dall'utente coinvolgano dispositivi fisici. Ad esempio, nel caso in cui un utente creasse la regola *"Se sono le 8 del mattino (Date&Time) allora accendi le luci (Philiphs Hue) della camera"*, il costo di esecuzione consisterebbe nell'aspettare le 8 e controllare che la porta della camera sia stata aperta correttamente.

Questa ovviamente non è una soluzione ottimale perché il processo di debug potrebbe avere durata potenzialmente infinita.

Per evitare questo problema si potrebbe creare un ambiente di simulazione dove l'esecuzione delle regole è rappresentata attraverso l'uso di alcune animazioni. In questo modo l'utente simulando le condizioni necessarie per l'attivazione della regola da lui definite potrà velocizzare il processo di debug.

Assistenza al Debug Per assistere l'utente durante la fase di debug si potrebbe utilizzare un implementazione dell'Interrogative Debugging supportato dall'uso di dataflow graph e spiegazioni testuali. Questo approccio potrebbe costituire una buona soluzione perché uno dei problemi maggiori che un utente si trova ad affrontare durante la programmazione di sistemi gestiti attraverso l'uso del Trigger-Action Programming, è dovuto all'interazione delle regole da lui definite. Questo tipo di problema è esattamente quello che in Alice2.0 è stato risolto grazie all'uso dell'Interrogative Debugging [17]. In questo modo è possibile comunicare in modo tempestivo ad un utente l'insorgere di errori causati dall'interazione con regole precedentemente definite (errori 2, 3, 4). Bisognerebbe inoltre creare dei meccanismi che permettano la modifica temporanea di una regola e il ripristino di questa per arginare i problemi dovuti ad *inflexibilty* e *self-confidence* di utenti non esperti.

Analisi delle regole eseguite Per aiutare ulteriormente un utente nel capire cosa abbia causato un comportamento indesiderato del sistema, potrebbe risultare utile progettare un modulo che sappia ricostruire le azioni eseguite dal sistema nelle stesse modalità usate dalla funzionalità del "what caused this to happen?" implementata dalla Whyline [17].

3.2 Mockup

In questa sezione verranno presentati i mockup realizzati durante la fase di design del sistema che costituiscono una delle possibili soluzioni per la progettazione di un tool di EUD per la composizione e il debug di regole trigger-action.

Utilizzando come punto di partenza quanto proposto nella sezione 3.1, sono state disegnate delle soluzioni, per una applicazione web, con le seguenti caratteristiche:

- composizione delle regole mediante l'uso block-programming.
- Rappresentazione di trigger e action attraverso la rappresentazione mediante puzzle.
- Controllo per gli errori come loop, inconsistenze e ridondanze in fase di composizione e rappresentazione di questi attraverso l'uso di colori, spiegazioni testuali e grafiche.
- Area di simulazione di regole ed ambiente per analisi delle regole eseguite.

La possibile soluzione ideata è composta da tre mockup principali:

1. Area di composizione.

   In questa area l'utente potrà comporre una regola e correggere tutti gli errori che causano la formazione di loop, ridondanze ed inconsistenze.

2. Area di simulazione.

   In questa area l'utente potrà simulare degli eventi e delle condizioni per verificare il corretto funzionamento delle regole da lui definite.

3. Area di analisi delle regole eseguite.

   In questa area l'utente potrà capire quali regole sono state eseguite dal sistema ed analizzarne il flusso di esecuzione.

Area di composizione della regola Una possibile soluzione che permetta ad un utente di comporre una regola è mostrata in figura 3.1.

In questa soluzione l'utente ha a disposizione una schermata composta da:

- lista di servizi disponibili tra i quali scegliere quello che offre il trigger o l'action desiderata.
- Un'area di lavoro dove trascinare i pezzi del puzzle agganciandoli tra loro .

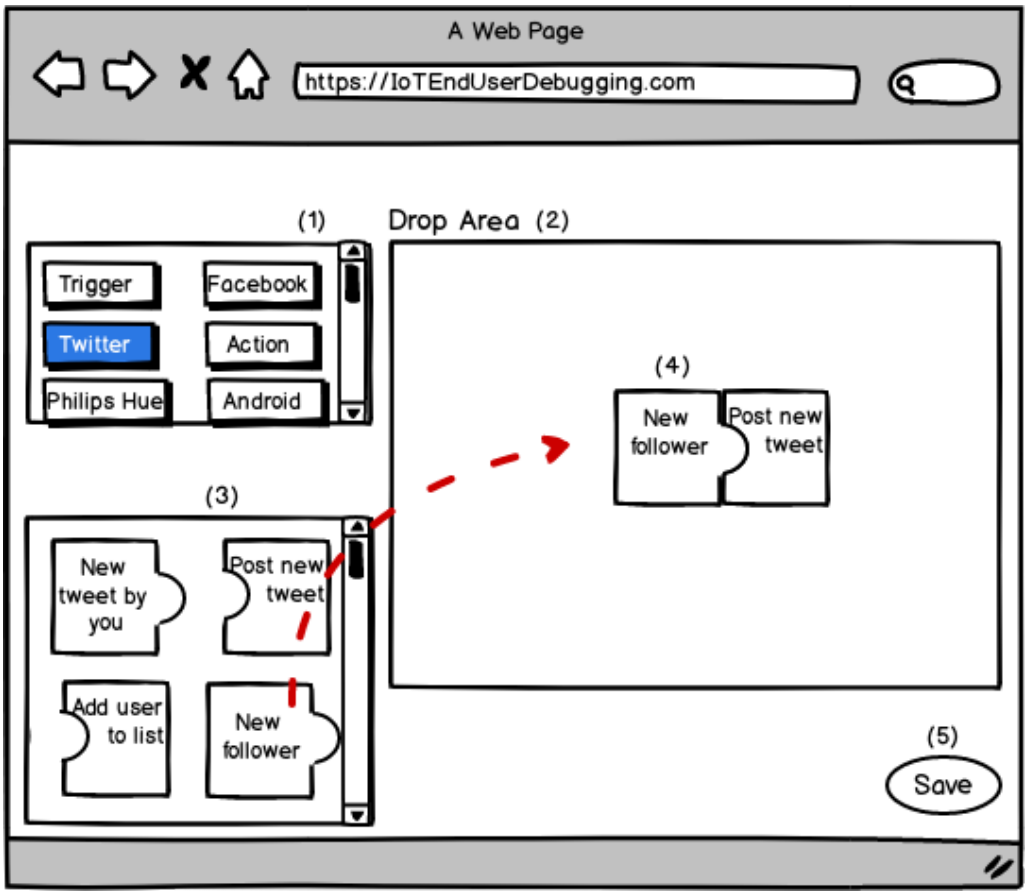


Figura 3.1: Area di composizione della regola. (1) Lista di servizi disponibili; (2) area di lavoro; (3) repository dei pezzi del puzzle; (4) nuova regola; (5) pulsante per salvare;

- Un repository in cui sono mostrati i pezzi del puzzle che dovranno essere utilizzati per comporre le regole.

- Un pulsante per salvare la regola.

Il processo di composizione di un regola, come già spiegato nella sezione 2.1, consiste nel trascinare i blocchi, aventi la forma di puzzle, dal repository all'area di lavoro. Quando due blocchi sono aggancianti tra di loro sarà creata una nuova regola e verrà verificato se la regola appena definita è in conflitto con quelle definite precedentemente.

Nel caso in cui non sia rilevato nessun problema l'utente potrà procedere con il salvataggio della regola, in caso contrario dovrà risolvere il problema notificato dall'interfaccia interagendo con la schermata mostrata in figura 3.2.

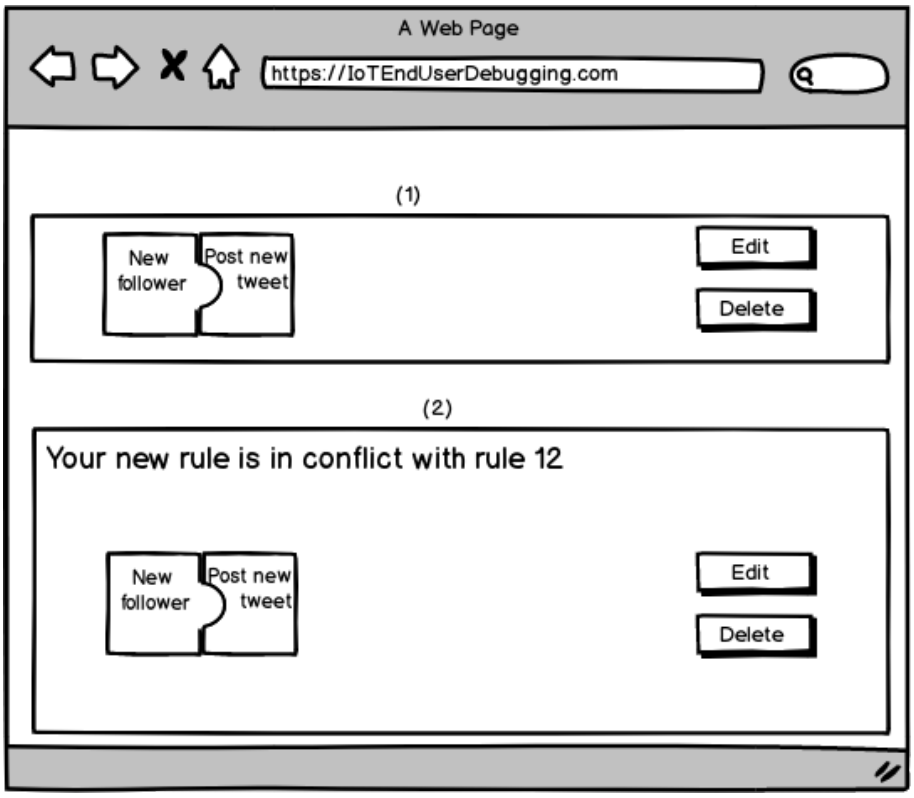


Figura 3.2: Area di risoluzione dei problemi. (1) Nuova regola; (2) regola salvata in precedenza

Per risolvere il problema rilevato, sarà possibile utilizzare i bottoni di "Edit" e "Delete" decidendo quale regola modificare o cancellare. Nel caso in cui l'utente decida di modificare una delle due regole sarà riportato alla schermata di composizione della regola selezionata.

Questa soluzione permette ad un utente di risolvere tutti i problemi dovuti all'interazione con le altre regole prima di salvare la regola creata evitando quindi di generare delle situazioni instabili che provocherebbero un comportamento anomalo del sistema.

Area di simulazione Come spiegato precedentemente la simulazione potrebbe costituire un'ottima soluzione per ridurre i "costi di esecuzione " che risultano essere molto alti in sistemi IoT che interagiscono con dispositivi fisici.

Una possibile soluzione può essere quella mostrata nel mockup in figura 3.3.

In questo caso la simulazione è composta da variabili ambientali, eventi e regole da testare. Il risultato è rappresentato sotto forma di animazioni. Per variabili

ambientali si intendono tutti i parametri esterni al sistema e rilevabili da sensori come ad esempio la temperatura di una camera oppure l'orario. Gli eventi, invece, rappresentano le condizioni che attivano un determinato trigger e scatenano quindi l'esecuzione di una regola.

In questa soluzione è possibile trovare:

- una serie di bottoni per definire variabili ambientali, eventi e regole da testare;
- lo spazio dove è riprodotta la simulazione grafica di ciò che accade al sistema;
- la lista di regole eseguite durante la simulazione. Per ogni regola sarà possibile avere maggiori informazioni sul motivo di esecuzione usando il paradigma di debug della Whyline nella variante del "Perché è successo l'evento X?".
- Uno spazio in cui è possibile vedere il cambiamento di alcune variabili relative alla regola che l'utente vuole testare;
- una linea temporale in cui sono distribuiti gli eventi definiti dall'utente;

Analisi real-time Un ulteriore aiuto che un tool di EUD potrebbe fornire ad un utente è quello dell'analisi delle regole eseguite dal sistema, in una determinata finestra temporale.

Una possibile soluzione è quella disegnata in figura 3.4. Questa soluzione permette all'utente di visualizzare tutte le regole eseguite nel lasso di tempo selezionato.

La rappresentazione del flusso d'esecuzione avviene tramite un dataflow associato ad una linea temporale. In questo dataflow tutte le regole eseguite in maniera concatenata sono unite da frecce in modo da rendere esplicito il legame che le lega. Anche in questo caso, come nel caso della simulazione, potrebbe essere una buona idea quella di associare ad ogni regola eseguita una spiegazione che fornisca le motivazioni principali che hanno portato alla sua esecuzione.

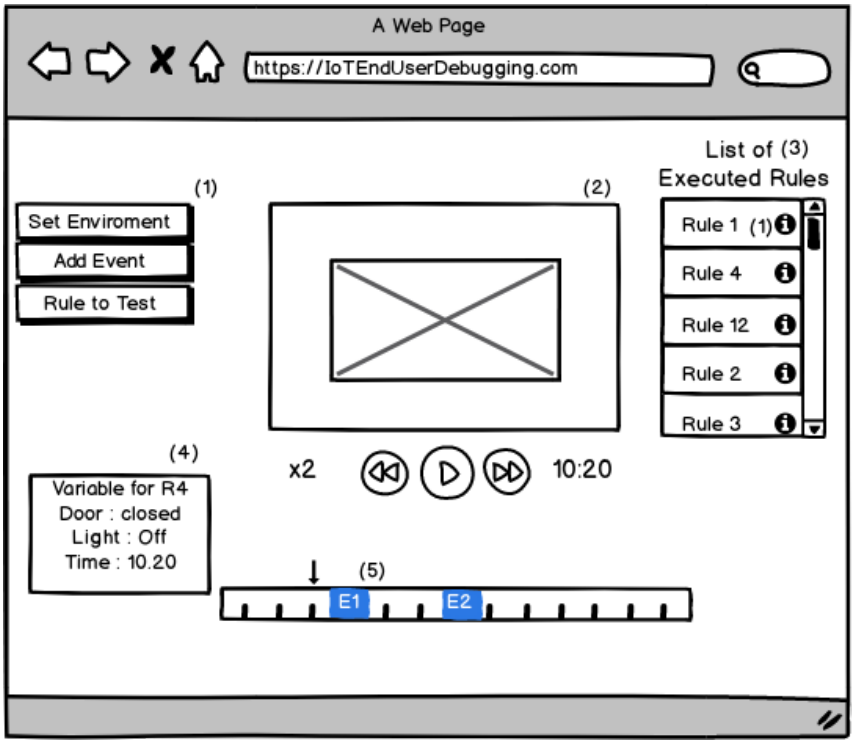


Figura 3.3: Area di simulazione. (1) Menu per settare i parametri della simulazione; (2) area per la rappresentazione grafica della simulazione; (3) lista di regole eseguite durante la simulazione; (4) variabili significative della regola da testare; (5) linea temporale

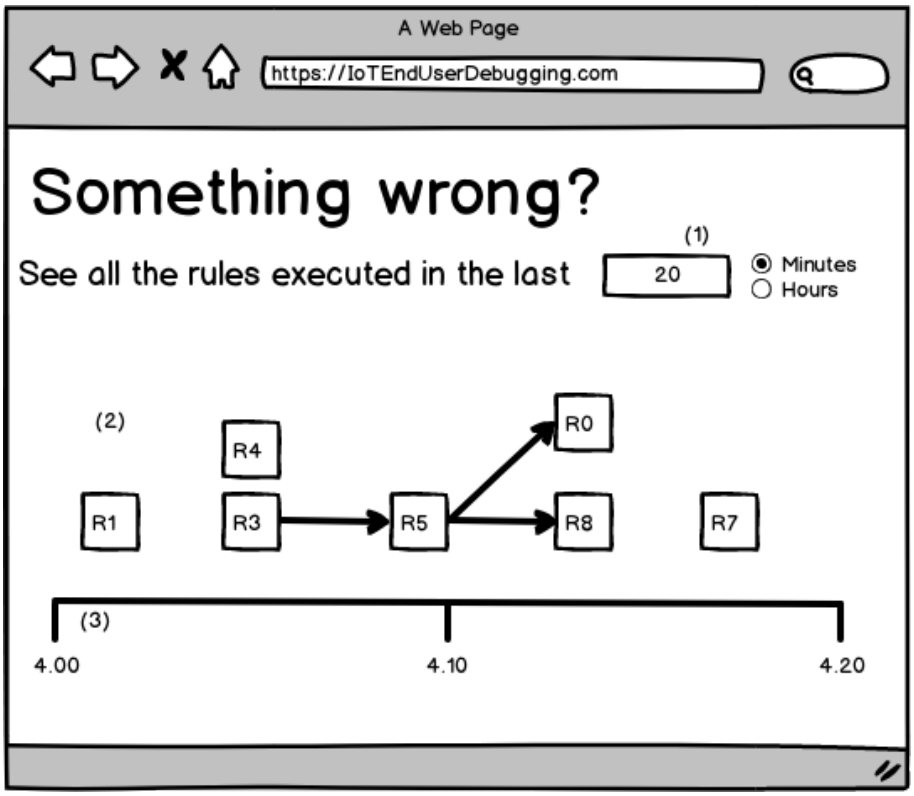


Figura 3.4: Area di analisi delle regole eseguite. (1) Dataflow; (2) linea temporale

3.3 Progettazione

A partire dai mockup generati, si è scelto di progettare un sistema che assista gli utenti nella fase di composizione e debug delle regole. Per questo motivo, la progettazione dell'interfaccia utente comprende l'area di composizione e di risoluzione dei relativi problemi (Figure 3.1, 3.2), lasciando per lavori futuri la simulazione e l'analisi delle regole eseguite.

L'interfaccia si focalizzerà, quindi, su come aiutare un utente a capire e risolvere i problemi generati dall'interazione con altre regole come loop, ridondanze ed inconsistenze. Il compito più importante del nostro tool consisterà nello spiegare in modo semplice ed intuitivo il concetto di loop, inconsistenza e ridondanza per permettere all'utente di capire quali siano le scelte giuste da compiere per risolvere i problemi generati.

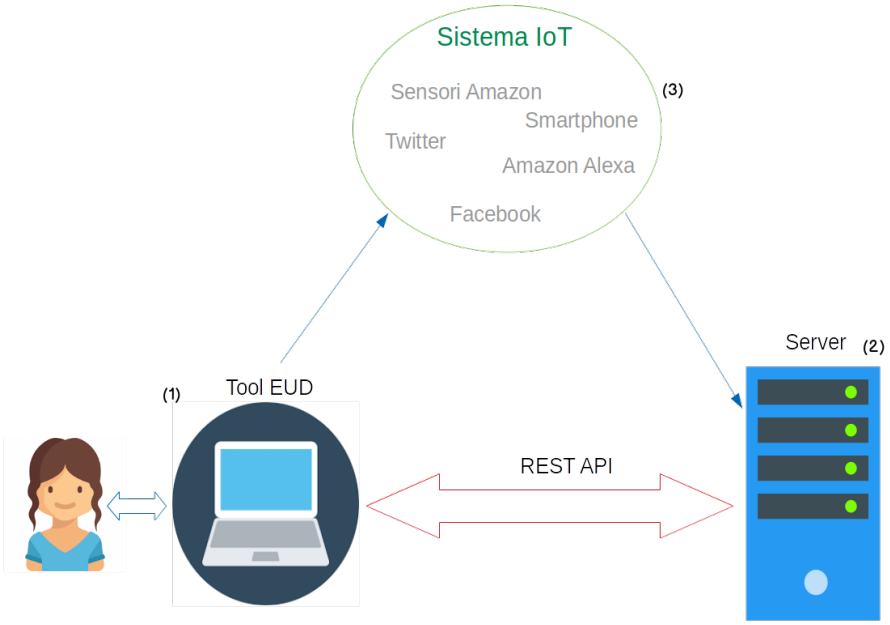


Figura 3.5: Architettura del sistema. (1) Tool per la creazione e debug di regole; (2) server REST; (3) sistema di dispositivi e servizi web creato dall'utente

L'infrastruttura generale del sistema è quella mostrata in figura 3.5. Un utente interagendo con il tool EUD potrà gestire il proprio ecosistema IoT creando delle regole e se necessario risolvere dei problemi. Il tool utilizzerà un server REST esterno, implementato precedentemente del gruppo e-Lite del Politecnico di Torino [12], che gestirà le regole create dall'utente.

Il compito del server REST è quello di memorizzare tutte le regole create dall'utente e rilevare, attraverso l'utilizzo di speciali algoritmi, se e quali regole sono in conflitto con la regola che l'utente sta cercando di creare.

Il server memorizzerà, inoltre, tutti i dispositivi registrati dall'utente che andranno a comporre quello che è definito come ecosistema IoT.

Un utente potrà creare ed effettuare il debug delle proprie regole seguendo il processo mostrato in figura 3.6.

L'utente interagendo con l'interfaccia grafica del nostro prototipo potrà quindi registrare nuovi dispositivi, creare regole e gestire le regole salvate in passato.

Inoltre in fase di composizione potrà capire quali sono i problemi che queste regole possono generare.

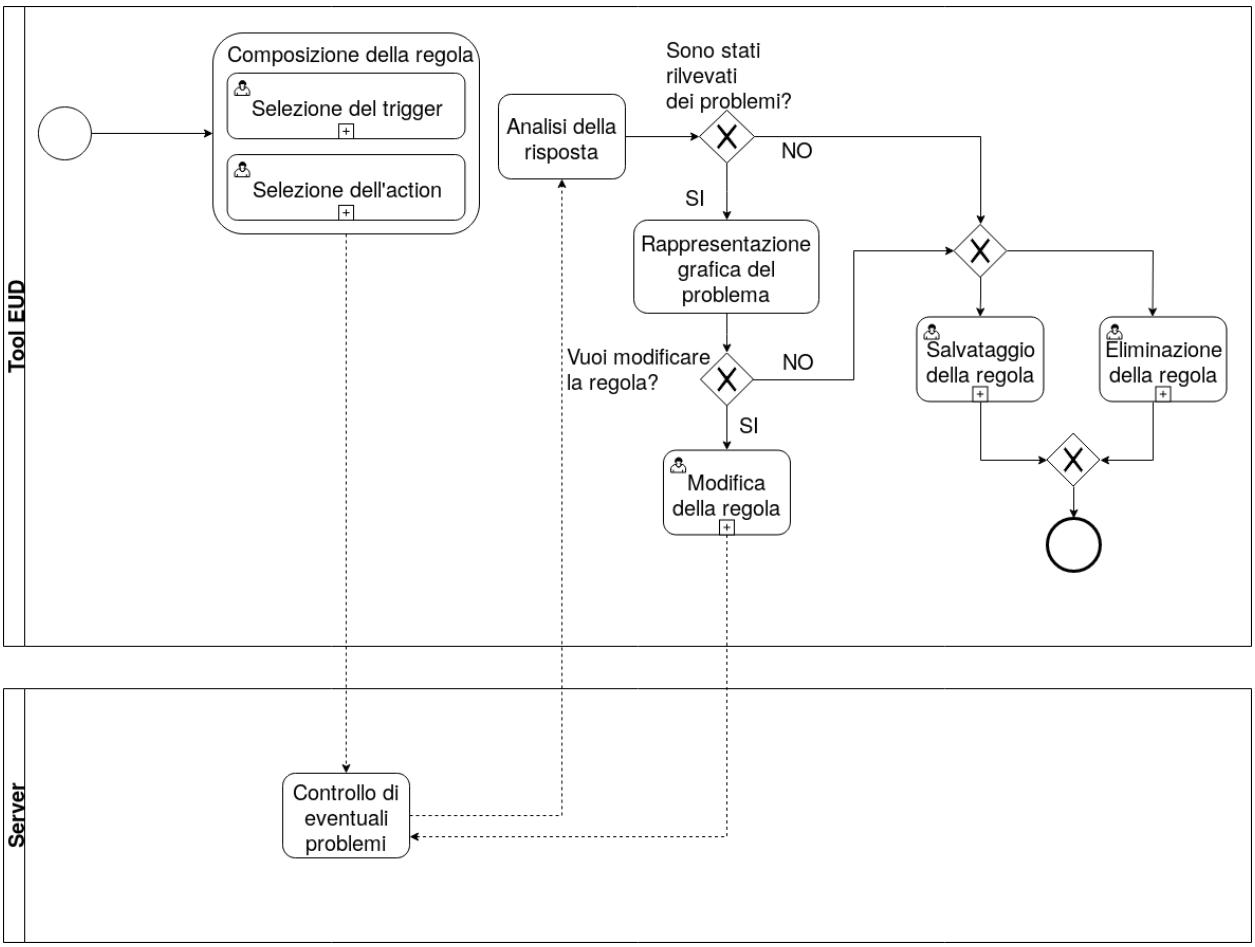


Figura 3.6: Diagramma BPMN del processo di composizione e debug

Per ragioni di semplicità è stato deciso che un utente possa comporre una sola regola alla volta e che una regola è formata da un solo trigger e una sola action.

Il tool EUD sarà composto da tre moduli principali che si occuperanno di:

- gestione del processo di composizione di una regola;
- segnalazione e rappresentazione di problemi come loop, inconsistenze e ridondanze;
- gestione delle regole salvate in passato.

L'area di composizione è stata progettata partendo dal mockup mostrato in figura 3.1. In questa area l'utente potrà comporre le regole desiderate trascinando i pezzi di puzzle, corrispondenti al trigger e all'action voluti, dal repository all'area di lavoro.

Già in fase di composizione l'interfaccia cercherà di fornire un primo feedback sulla scelta dei componenti da utilizzare. Per scoraggiare l'utente all'uso degli stessi pezzi di puzzle più volte, ad esempio, si è deciso di aggiungere delle colorazioni che vadano a simulare l'usura del pezzo stesso. L'utilizzo di uno stesso trigger associato ad azioni diverse, infatti, aumenta la probabilità di incorrere in errori dovuti all'interazione con altre regole.

Terminato il processo di composizione della regola il tool EUD, grazie a delle interazioni con il server REST, comunicherà all'utente se quella regola genera o meno dei problemi. Si cercherà inoltre di comunicare la gravità del problema generato. Loop ed inconsistenze sono considerati problemi gravi e a loro è associato il colore rosso, questi sono trattati come veri e propri errori che potrebbero rendere il sistema instabile. Le ridondanze invece sono considerate errori non gravi e quindi sono associati al colore giallo. In caso di salvataggio, in questo caso, il sistema non sarebbe instabile ma potrebbe eseguire più volte azioni che portano il sistema allo stesso stato.

Nel caso in cui siano rilevati dei problemi il prototipo condurrà l'utente verso l'area creata per la risoluzione dei conflitti (Figura 3.2).

Indipendentemente dai problemi rilevati dal sistema l'utente potrà decidere in ogni momento di salvare o eliminare la regola creata.

In questa area l'utente troverà varie rappresentazioni che lo possano aiutare a capire il problema segnalato e dovrà decidere se salvare, modificare o eliminare la regola creata.

Il prototipo consentirà anche di visualizzare tutte le regole precedentemente salvate ed eliminare quelle selezionate dall'utente.

Capitolo 4

Implementazione

In questo capitolo verranno illustrate quali sono state le tecnologie e le tecniche utilizzate in fase di implementazione del prototipo. Successivamente sarà illustrato il funzionamento passo-passo dell'interfaccia creata.

4.1 Dettagli implementativi

Il sistema è composto da un'architettura client-server. Il client è il tool EUD creato in questa tesi ed è un'applicazione web. Il server è un server REST implementato precedentemente del gruppo e-Lite del Politecnico di Torino [12]. I linguaggi di programmazione utilizzati per l'implementazione dell'applicazione web sono: HTML, CSS e Javascript. Sono state, inoltre, utilizzate le librerie W3.CSS[1] e JQuery[2].

La maggior parte del codice html è generato automaticamente dai componenti javascript in base agli input forniti dall'utente e i dati ricevuti in seguito a delle interazioni con il server. È stato inoltre deciso di implementare tutta l'interfaccia grafica in inglese. Le rappresentazioni grafiche sono state create esclusivamente attraverso l'utilizzo congiunto di Javascript e CSS. Le funzionalità dell'applicazione web sono state implementata con Javascript.

4.1.1 Implementazione del client

Il tool implementato è diviso in più componenti che gestiscono operazioni diverse comunicando tra loro attraverso l'utilizzo di strutture dati condivise. I moduli principali che implementano l'interfaccia si occupano di:

[1] "https://www.w3schools.com/w3css/4/w3.css", ultimo accesso il 07/10/2018

[2] "https://code.jquery.com/ui/1.12.1/jquery-ui.js", ultimo accesso il 07/10/2018

- gestire le strutture dati usate;
- gestire le componenti grafiche che fanno uso di meccanismi drag&drop. In questo modulo è stato fatto largo uso delle funzioni implementate dalla libreria jQuery.
- Gestire le regole create;
- gestire la comunicazione con il server REST;
- creare i grafici utilizzati nell'area di risoluzione dei conflitti.

Tutti gli elementi grafici con cui è possibile interagire sono dotati di un ID univoco attraverso il quale è possibile identificare la struttura dati che lo descrive. L'uso di queste strutture dati ha consentito di gestire in maniera semplice tutti i problemi di interazione e sincronizzazione necessaria tra i vari moduli del sistema.

Le interazioni con il server avvengono attraverso l'uso delle API messe a disposizione dal server e le informazioni sono trasmesse in formato JSON.

Il server implementa una rete di Petri per rilevare loop, inconsistenze e ridondanze e per ognuno di questi problemi fornisce le informazioni necessarie per descriverlo. Le informazioni necessarie per descrivere i problemi conterranno:

- trigger iniziale che scatena l'esecuzione del set di regole ridondanti o inconsistenti
- lista di regole ridondanti o inconsistenti. Ogni regola ha un attributo "parent" che rappresenta la regola da cui è stata attivata. Percorrendo l'albero dei "parent", in modo ricorsivo, è possibile ricreare una struttura ad albero che vada a rappresentare il flusso di esecuzione eseguito dalla simulazione.
- Lista di regole eseguite

Il tool di EUD deve creare la catena di esecuzione, composta dall'insieme delle regole "parent", per ogni regola che risulti essere problematica. In questo modo per ogni regola, marcata come problema, sarà possibile ricavare tutte le regole che hanno portato alla sua esecuzione utilizzando come punto di partenza il trigger iniziale specificato. È possibile che la catena di esecuzione di una regola sia una sotto-catena dell'esecuzione di un'altra regola. In questi casi per avere un grafico più nitido tutte le sotto-catene saranno rimosse.

4.1.2 Interazione con il server

Le richieste effettuate dal tool al server sono di due tipo: GET e POST. Queste richieste sono effettuate in maniera asincrona e per questo motivo è necessario implementare dei meccanismi di sincronizzazione che permettano ai moduli del tool

di gestire correttamente le informazioni ricevute. Questo processo è sincronizzato attraverso l'uso di flag, variabili condivise e meccanismi di lock.

L'utente prima di poter utilizzare il tool di EUD verrà identificato attraverso un login implementato con il protocollo di HTTP Basic Authentication.

4.2 Funzionamento dell'interfaccia finale

Questa sezione descrive le funzionalità più importanti del sistema implementato attraverso un'esecuzione passo-passo del processo di composizione e debug che un utente potrebbe seguire.

4.2.1 Area di composizione

L'interfaccia iniziale del nostro prototipo è quella mostrata in figura 4.1.

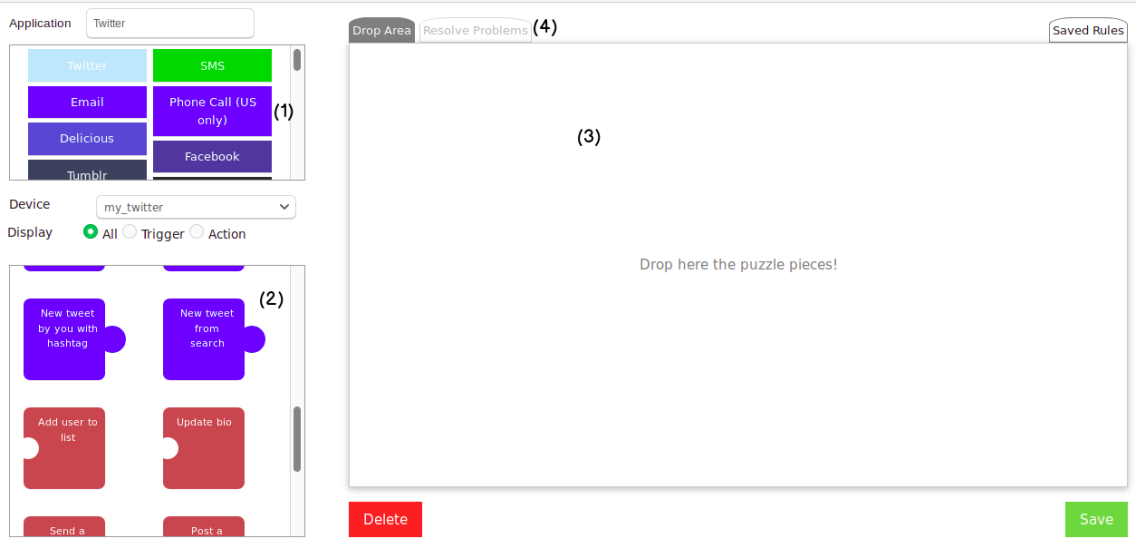


Figura 4.1: Area di composizione. (1) Lista di servizi; (2) repository; (3) area di lavoro; (4) menu da utilizzare per cambiare area;

Questa interfaccia non si discosta di molto da quella disegnata in fase di mockup (Figura 3.1) e è composta da:

- un'area dove sono contenute tutte le applicazioni
- un repository dove sono mostrati trigger e action sotto forma di puzzle
- un'area di composizione della regola

Ogni regola composta è formata da un trigger e una action. Ad ogni trigger, ed action, è associato sia il servizio da cui questo è offerto , ad esempio Philips Hue, sia il canale attraverso cui questo viene fornito, ad esempio lampada da cucina.

Inoltre ogni trigger, o action, può avere informazioni aggiuntive specifiche.

Tutte le informazioni su servizi, canali e informazioni aggiuntive sono inviate dal server REST al tool su richiesta.

Il sistema realizzato chiederà in fase di composizione all'utente di inserire tutte le informazioni necessarie, sia per il trigger che per l'action, affinché la regola composta sia corretta.

Questo avviene in quattro passi, che dovranno essere ripetuti sia per la selezione del trigger che per la selezione dell'action:

1. selezione del servizio, fisico o virtuale, desiderato;

2. selezione del canale attraverso cui il servizio è offerto.

3. Scelta del puzzle e trasporto di questo dal repository all'area di lavoro attraverso un meccanismo basato sul drag&drop;

4. in seguito al rilascio del puzzle all'interno dell'area di lavoro, inserimento di tutte le informazioni aggiuntive richieste dall'interfaccia;

I servizi sono rappresentati da blocchi con forma rettangolare nell'apposita area come rappresentato in figura 4.2. Per facilitare la selezione del servizio desiderato è possibile utilizzare la barra di ricerca apposita.

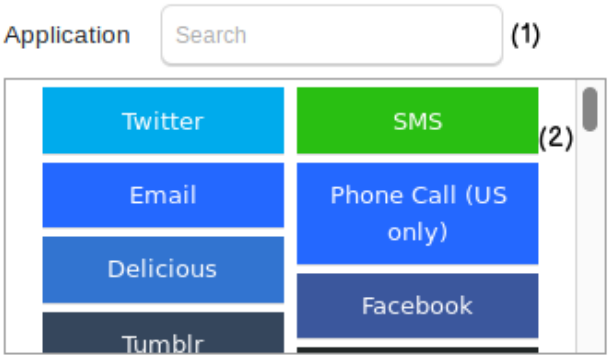
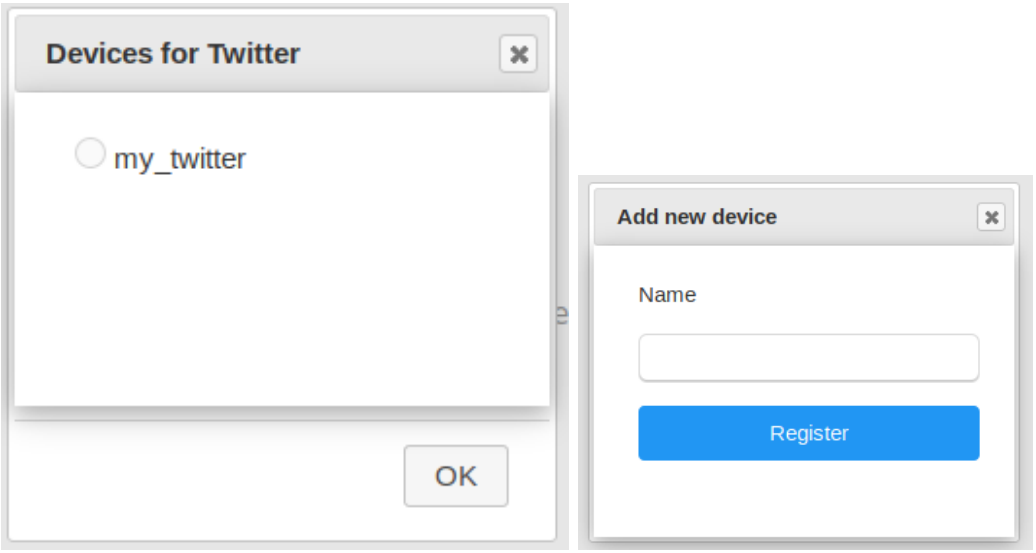


Figura 4.2: (1) Barra di ricerca; (2) lista di servizi

Dopo aver selezionato il servizio desiderato l'utente dovrà decidere quale canale utilizzare per fornire il servizio, nel caso in cui non ci sia nessun canale registrato l'utente dovrà registrarne almeno uno (figura 4.3).



(a) selezione di un canale tra quelli registrati (b) registrazione di un nuovo canale

Figura 4.3: Procedura di selezione del canale

È possibile cambiare in ogni momento il canale che fornisce il servizio o il servizio stesso ripetendo i passi 1 e 2 o il alternativa utilizzando il menu apposito mostrato in figura 4.4 contenente tutti i possibili canali associati a quello specifico servizio.

Dopo che un'utente avrà selezionato il servizio da utilizzare, con il relativo canale, il tool mostrerà nel repository tutti i trigger e le action relative all'applicazione selezionata come mostrato in figura 4.4.

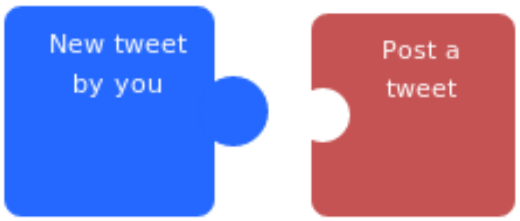
Un esempio di trigger ed action è quello mostrato in figura 4.5.

Un trigger si differenzia da un'action per forma e colore. Le forme, nel rispetto del paradigma del block-programming con rappresentazione mediante puzzle, risultano essere complementari, i colori al contrario sono diversi. Il colore non dipende solo dal tipo del puzzle, trigger o action, ma anche dall'utilizzo che se ne è fatto. Il colore, in particolare, rifletterà lo "stato di usura" di un pezzo di puzzle, ovvero il numero di volte in cui tale pezzo è già stato utilizzato. Il numero di utilizzi del puzzle verrà aggiornato dopo ogni salvataggio di una regola. Il colore cambierà quindi in tempo reale. Gli stati di usura, con relative colorazioni, implementate nel prototipo sono tre: assente, lieve, grave. Un esempio può essere quello mostrato in figura 4.6.

La forma e la grandezza del pezzo del puzzle, indipendentemente dal tipo, non permette di inserire dei titoli molto precisi al suo interno. Per questo si è deciso di aggiungere delle descrizioni, mostrate all'utente sotto forma di tooltip, come mostrato in figura 4.7.

Una volta selezionato l'elemento desiderato l'utente potrà trascinarlo dal repository all'area di lavoro.

Figura 4.4: (1) Menu per selezionare il canale; (2) menu per selezionare cosa visualizzare nella repository; (3) repository.



(a) trigger (b) action

Figura 4.5: Esempi di pezzi di puzzle

(a) assente (b) lieve (c) grave

Figura 4.6: Stati di usura

Il tool consentirà ad un utente di depositare il pezzo nell'area di lavoro solo nel caso in cui nel piano di lavoro non sia già presente una regola oppure un elemento dello stesso tipo. Nel caso in cui queste condizioni non siano verificate apparirà un

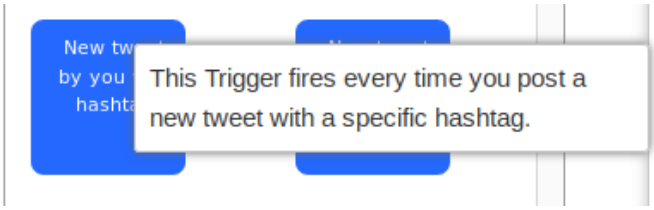


Figura 4.7: Area di risoluzione dei problemi. (1) Nuova regola; (2) Regola salvata in precedenza

messaggio di errore che informerà l'utente di questo problema. In questo modo si eviteranno tutti i possibili errori sintattici nella formazione di una regola.

Nel caso in cui vada tutto a buon fine l'utente riuscirà a rilasciare il pezzo scelto nell'area di lavoro. A questo punto, se necessario, dovrà inserire tutte le informazioni aggiuntive richieste dal componente selezionato come mostrato in figura 4.8.

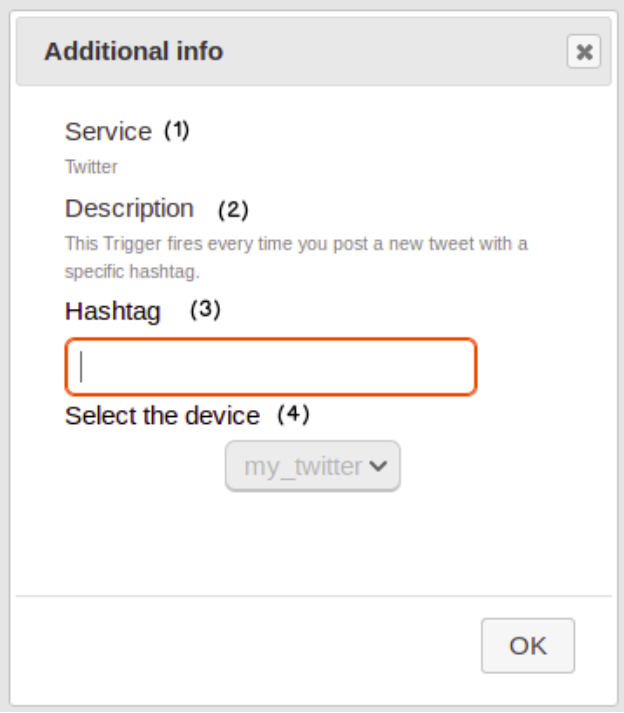


Figura 4.8: (1) Servizio che offre l'elemento; (2) breve descrizione dell'elemento; (3) informazioni aggiuntive; (4) canale attraverso cui è offerto il servizio

Nel caso in cui l'utente non inserisca le informazioni necessarie oppure cerchi di rilasciare il puzzle in un area non idonea il pezzo sarà eliminato dall'area di lavoro.

L'utente può decidere di eliminare in ogni momento una o più componenti della regola trascinandole nell'area apposita come mostrato in figura 4.9.

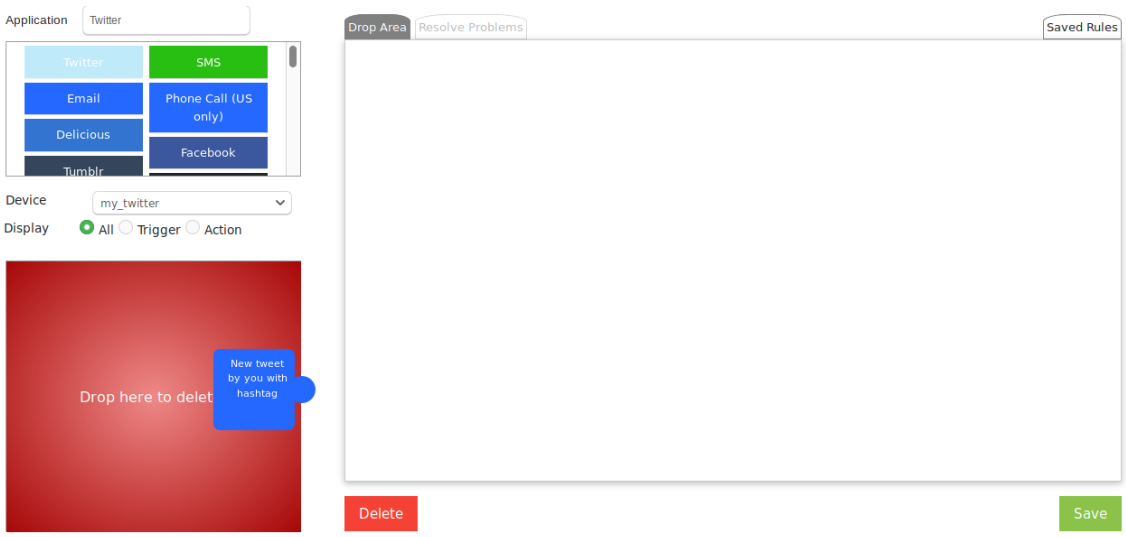


Figura 4.9: Eliminazione di un elemento dall'area di lavoro.

Quando un'utente terminerà di comporre la propria regola, l'interfaccia interagirà con il server per capire se la nuova regola genera dei problemi con le regole precedentemente salvate.

La fase di controllo verrà visualizzata dall'utente come mostrato in figura 4.10. Durante il controllo l'utente non potrà modificare alcun parametro della regola formata.

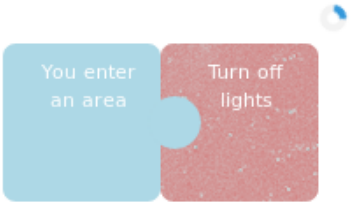


Figura 4.10: Rappresentazione in attesa di risposta del server

Le possibili risposte del server possono essere di tre tipi:

1. Non è stato rilevato nessun problema.

2. È stato rilevato almeno un problema grave. Sono classificati come problemi gravi i loop e le inconsistenze.

3. È stato rilevato almeno un problema ma non è grave. È classificato come problema non grave la ridondanza.

La risposta del server verrà rappresentata, dall'interfaccia, attraverso due rappresentazioni diverse: testuale e grafica. Per ogni tipo di rappresentazione esistono tre varianti diverse che rappresentano le tre possibili risposte del server (Figura 4.12).

Nel caso in cui ci sia stato un problema, grave o non grave, l'utente potrà accedere all'area di risoluzione dei problemi, descritta nel prossimo paragrafo, selezionando l'icona posta sulla regola stessa come mostrato in figura 4.11.

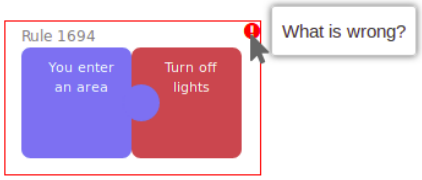
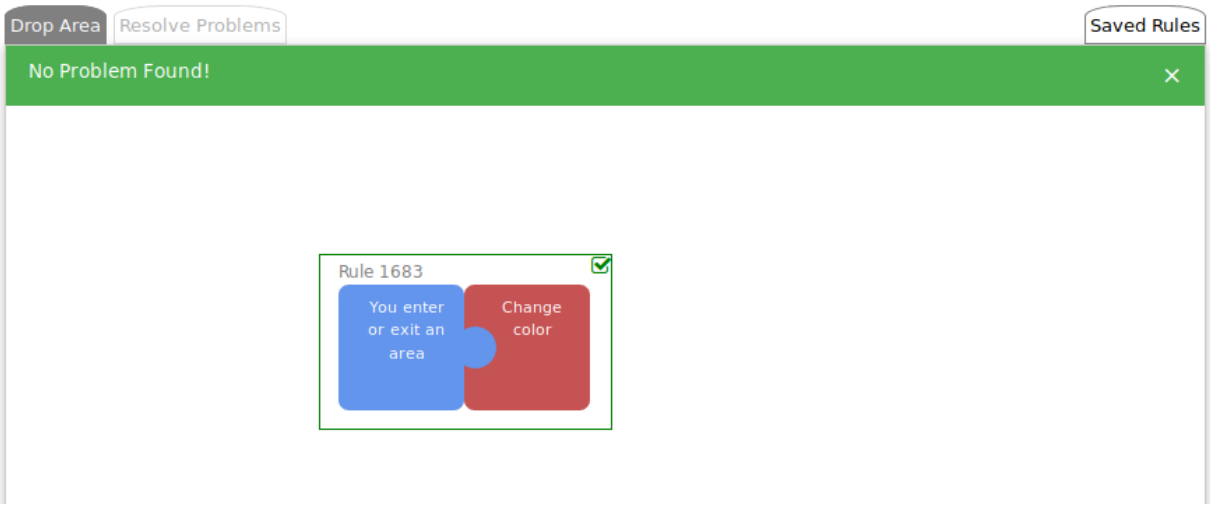


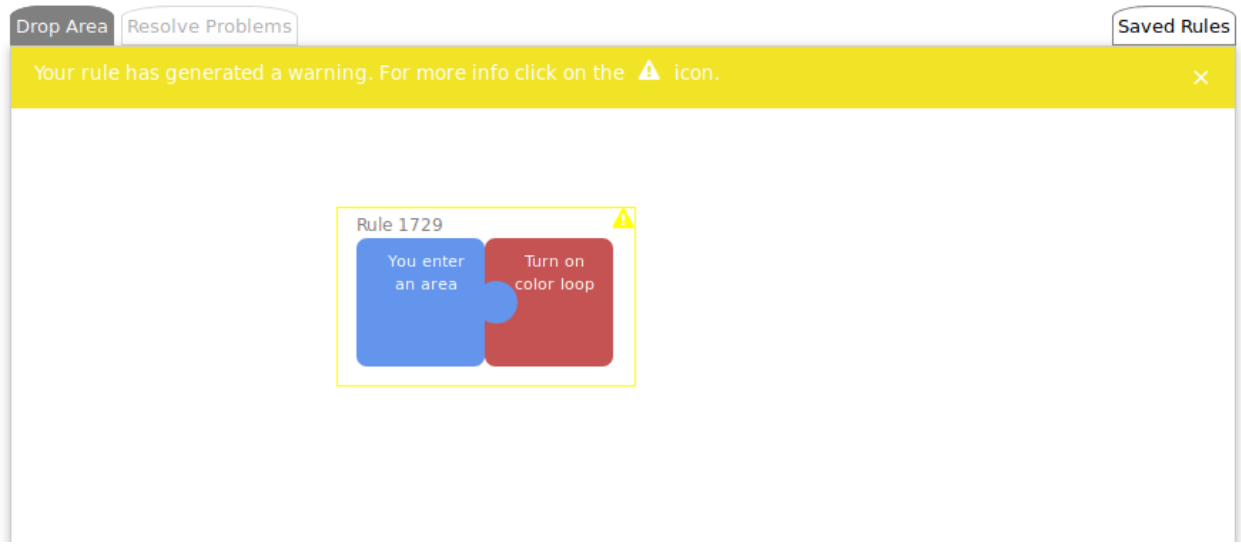
Figura 4.11: Accesso all'area di risoluzione di problemi.

Indipendentemente dalla risposta del server l'utente potrà decidere di salvare la regola, modificarne una componente o eliminare tutta la regola composta.

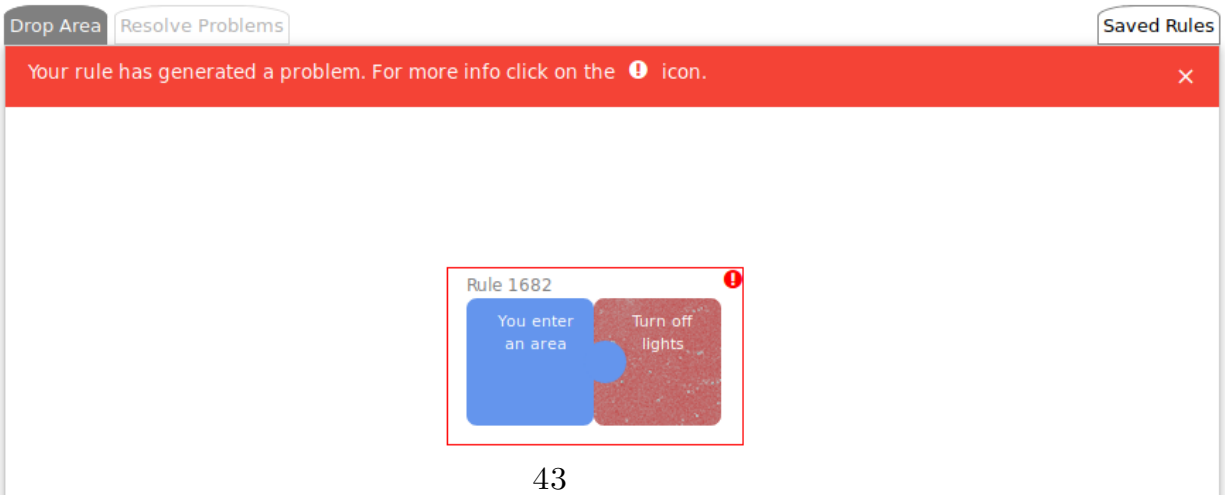
In questa versione si è deciso di permettere agli utenti di poter salvare le regole anche nel caso in cui queste possano causare problemi come loop, inconsistenze e ridondanze. Questo è stato fatto perché la simulazione effettuata dal server non prende in considerazioni tutti i parametri ma solo alcuni. Questo vuol dire che in alcuni casi il problema rilevato in realtà potrebbe non verificarsi. In alcuni casi, come ad esempio nel caso delle ridondanze, l'utente potrebbe volere comunque compiere quell'azione anche se questa logicamente risulti essere ridondante.



(a) Nessun problema rilevato



(b) Rilevato almeno un problema non grave



(c) Rilevato almeno un problema grave

Figura 4.12: Rappresentazione grafica della risposta del server

4.2.2 Area di Risoluzioni problemi

L'area di risoluzione dei problemi è accessibile solo nel caso in cui ci sia sul piano di lavoro una regola che abbia generato dei conflitti.

La prima implementazione di questa area seguiva lo sketch realizzato in fase di mockup. Il risultato è quello visibile in figura 4.13.

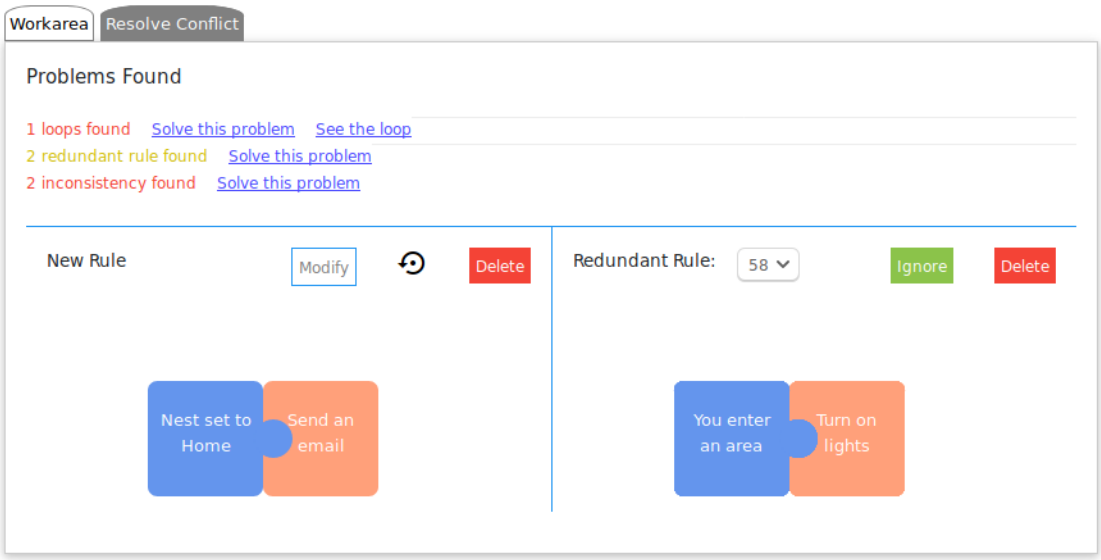


Figura 4.13: Prima implementazione per la rappresentazione del problema

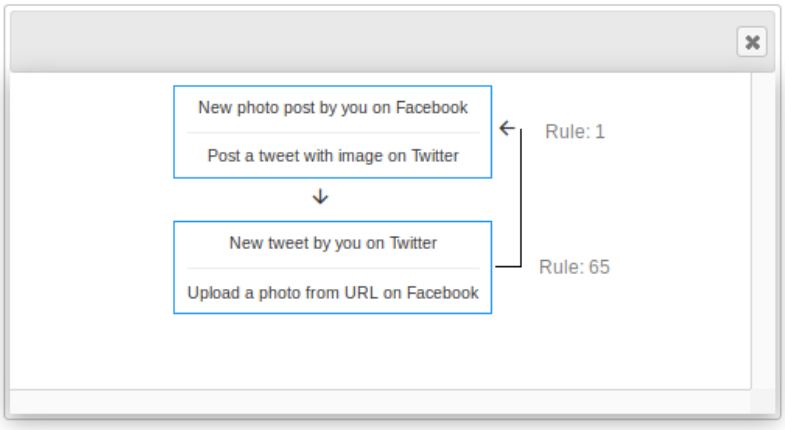


Figura 4.14: Prima implementazione per la rappresentazione di un loop

In seguito ad una analisi interna è stato deciso che questa soluzione risultava essere poco chiara nonostante i diagrammi utilizzati per spiegare all'utente il concetto di loop, inconsistenza e ridondanza, come quello visibile in figura 4.14.

È stato deciso che l'usabilità di questa interfaccia fosse troppo bassa rispetto a quella voluta per diverse ragioni:

- i diagrammi non erano visualizzati nella schermata principale ma visibili solo su esplicita richiesta dell'utente;
- non era presente nessuna spiegazione testuale simile a quelle usate dall'Interrogative Debugging;
- l'interfaccia non riusciva a guidare in maniera efficiente l'utente nel processo di debug.

Per risolvere questi problemi si è deciso di progettare ed implementare una seconda soluzione mostrata in figura 4.15.

Durante la fase di progettazione si è deciso che in questa seconda soluzione si dovesse:

- introdurre una spiegazione testuale;
- introdurre un dataflow più chiaro e renderlo sempre visibile;
- consentire all'utente di modifica o eliminare solo la regola da lui creata in fase di composizione.

In questo caso il risultato ottenuto è stato molto più soddisfacente sia in termini di chiarezza che di usabilità.

L'accesso a questa area avviene come mostrato nella sotto-sezione 4.2.1. Inizialmente quest'area risulterà essere disabilitata e solo nel momento in cui l'utente creerà una regola problematica verrà attivata. Una volta attivata l'utente potrà muoversi liberamente tra tutte le aree del tool attraverso l'uso del menu posizionato nella parte superiore della finestra (figura 4.16).

In ogni momento sarà possibile salvare o eliminare la regola creata. Nel momento in cui la regola composta sarà salvata o eliminata l'area di risoluzione di conflitti verrà disabilitata e le uniche aree attive saranno quella di composizione e quella di visualizzazione delle regole salvate.

Questa implementazione della finestra di risoluzione conflitti è divisa in tre aree:

1. area testuale dove è implementata la versione di Interrogative Debugging del tipo "Perché è successo l'evento X?";
2. area di visualizzazione del flusso di esecuzione mediante rappresentazione con dataflow;
3. area di modifica della regola.

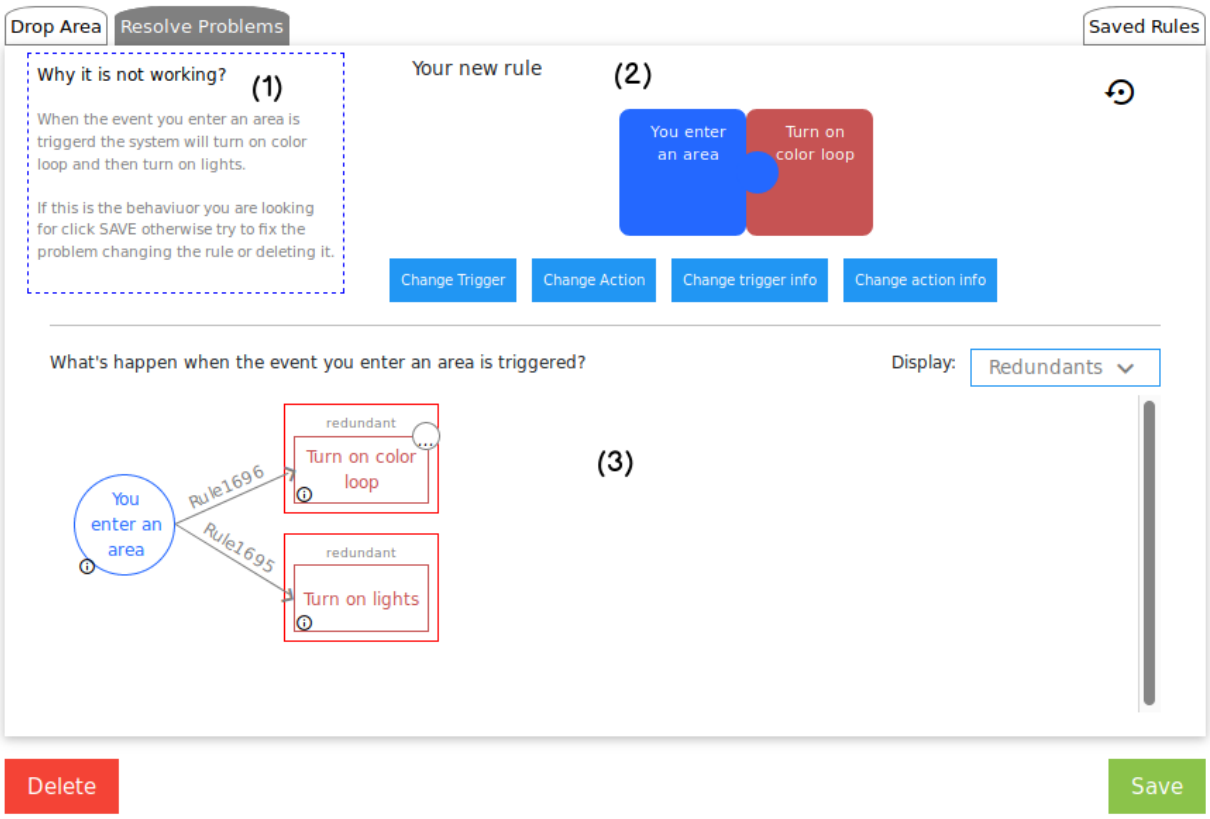


Figura 4.15: Area di risoluzione dei conflitti. (1) Spiegazione testuale; (2) regola modificabile; (3) spiegazione grafica.

Figura 4.16: Menu da utilizzare per muoversi tra le varie aree

Area testuale L'obiettivo dell'area testuale è quella di spiegare attraverso l'utilizzo di un linguaggio testuale il problema rilevato dall'interfaccia. Questo è fatto descrivendo cosa accade nel sistema attraverso l'uso di parole chiavi. L'area testuale è stata creata perché, da quanto si evince in letteratura, la rappresentazione mediante grafici non risulta essere sempre la rappresentazione più chiara per tutti utenti ma che alcuni preferiscano leggere un testo che descriva il flusso d'esecuzione. Questo tipo di approccio è largamente utilizzato nell'Interrogative Debugging [17].

Le spiegazioni testuali implementate dal tool tentano di mettere in risalto tre elementi:

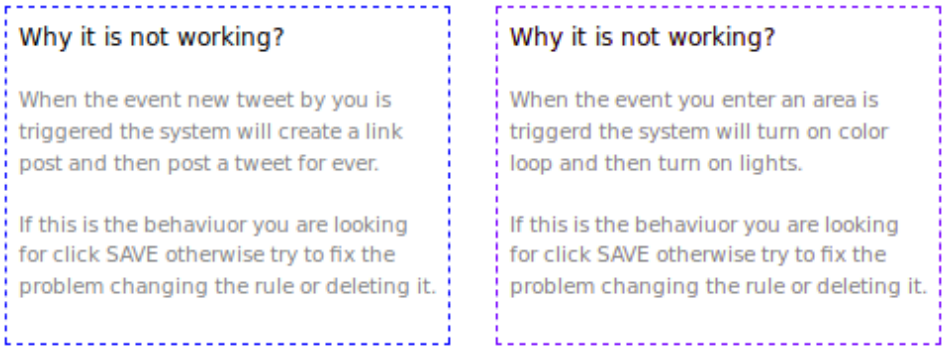
1. Il trigger che scatena il set di eventi che creano loop, ridondanze o inconsistenze. È importante che l'utente capisca quale sia la condizione effettiva di attivazione perché questa può differenziarsi da quella utilizzata nella creazione della regola che ha generato dei problemi.

2. L'insieme di azioni eseguite, dal sistema, in seguito all'attivazione del trigger che le scatena.

3. Parole chiave che rendano chiara la spiegazione di quale siano le conseguenze causate dal problema generato. Queste parole chiavi sono, ad esempio, nel caso di loop, "per sempre".

4. Le azioni che un utente può compiere per risolvere il problema.

In figura 4.17 è possibile trovare degli esempi di area testuale nel caso di loop, ridondanza.



(a) Loop (b) Ridondanza

Figura 4.17: Rappresentazione testuale

Rappresentazione grafica La seconda area è quella dedicata alla rappresentazione grafica del problema mediante dataflow.

Si è deciso di utilizzare il dataflow perché, dallo studio in letteratura, è emerso che questo tipo di rappresentazione risulta essere la più chiara nel caso in cui si debbano rappresentare dei flussi di esecuzione mettendone in risalto le connessioni.

Per tipo di rappresentazione si è utilizzata una notazione diversa per rappresentare trigger ed action rispetto a quella utilizzata in fase di composizione. I trigger e le action saranno rappresentate come mostrato in figura 4.18. È stato deciso di utilizzare gli stessi colori, utilizzati nella rappresentazione mediante puzzle, e di

inserire un tasto di informazioni (“i”) per poter avere maggiori informazioni sul quel determinato elemento come il servizio da cui offerto e le informazioni aggiuntive che lo caratterizzano.

(a) Trigger (b) Action

Figura 4.18: Elementi del dataflow

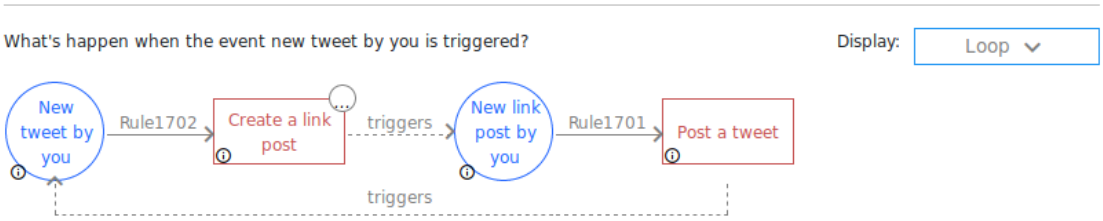
Un dataflow sarà composto da elementi, trigger o action, e connessioni. Le connessioni possono essere di due tipi: dal trigger ad un action e dall’action a ad un trigger. Nel primo caso quella connessione rappresenta l’esecuzione di una regola, nel secondo caso invece, è un azione che implicitamente crea delle condizioni che provocano lo scatenarsi di un trigger. Queste due tipologie di frecce sono rappresentate in modo diverso tramite l’uso di una linea diversa e una descrizione diversa come mostrato dalle figure 4.19, 4.20.

Figura 4.19: Connesione da action a trigger

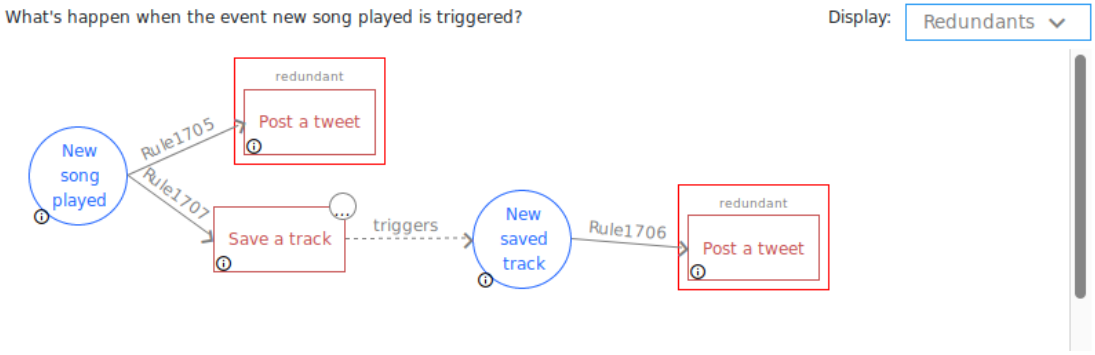
Figura 4.20: Connessione da trigger a action

La struttura del dataflow dipende strettamente dal il flusso di esecuzione che ha causato il problema rilevato.

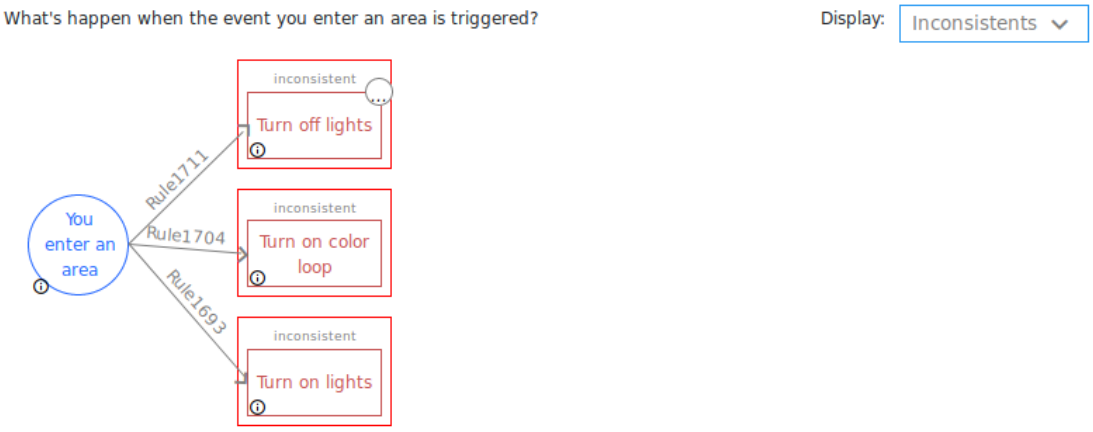
In caso di ridondanza e inconsistenza è possibile definire quale siano le azioni specifiche in conflitto. In questo caso le azioni saranno evidenziate attraverso l’uso di un riquadro rosso e un testo che ne dichiara il tipo di conflitto. Questo non sarà fatto in caso di loop perché per definizione tutte le regole che fanno parte del loop sono causa del conflitto. Un esempio di dataflow in caso di ridondanze, inconsistenze e loop è mostrano in figura 4.21.



(a) Esempio di rappresentazione grafica in caso di loop



(b) Esempio di rappresentazione grafica in caso di ridondanza



(c) Esempio di rappresentazione grafica in caso di inconsistenza

Figura 4.21: Esempi di dataflow utilizzati per la rappresentazione grafica del problema

Nel caso in cui siano presenti più problemi l'utente potrà decidere quale visualizzare di volta in volta attraverso l'uso di un dropdown menu mostrato in figura 4.22. Ogni qual volta l'utente cambierà il problema da visualizzare tutte le aree si

aggiorneranno mostrando le rappresentazioni, testuale o grafiche, corrette.

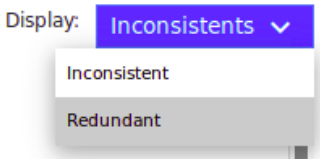


Figura 4.22: Menu per selezionare il problema da visualizzare e risolvere

Area di modifica La terza area è quella dedicata alla modifica della regola creata dall'utente in fase di composizione ed è mostrata in figura 4.23.

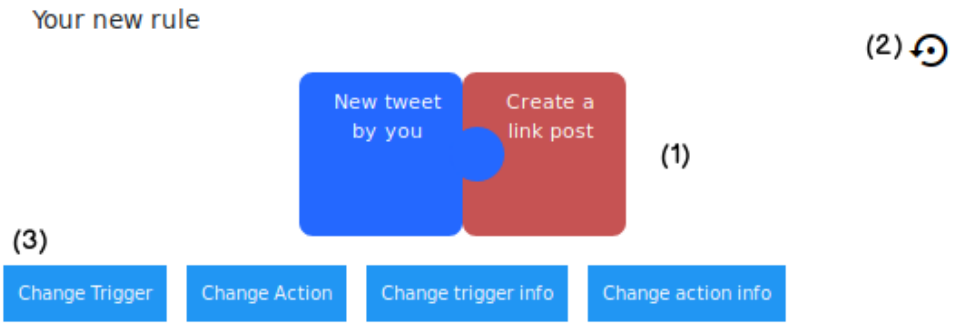


Figura 4.23: Area di modifica. (1) Regola modificabile; (2) bottone per il ripristino delle condizioni iniziali; (3) bottoni per la modifica della regola.

In questo caso la regola sarà rappresentata attraverso l'uso della puzzle metaphor, come avviene in fase di composizione.

In questa area l'utente potrà cercare di risolvere il problema rilevato dal server modificando la regola creata. Potrà fare ciò utilizzando quattro bottoni diversi che permettono di:

- modificare il trigger che compone la regola;
- modificare l'action che compone la regola;
- modificare le informazioni aggiuntive del trigger che compone la regola;
- modificare le informazioni aggiuntive dell'action che compone la regola;

Si è deciso di permettere ad un utente di modificare solo un elemento tra trigger ed action. Questo perché modificare entrambe le componenti corrisponderebbe alla

creazione di una regola completamente nuova e questo non è quello l'obiettivo da raggiungere in questa area.

L'utente ha, inoltre, a disposizione un pulsante che permette di ripristinare le specifiche iniziali della regola.

Nel caso in cui l'utente decida di modificare il trigger o l'action, il prototipo creerà una nuova finestra sullo schermo. L'utente, utilizzando lo stesso meccanismo usato per la creazione della regola nell'area di composizione, potrà selezionare un nuovo pezzo e trascinarlo nella finestra apposita, come mostrato in figura 4.24.

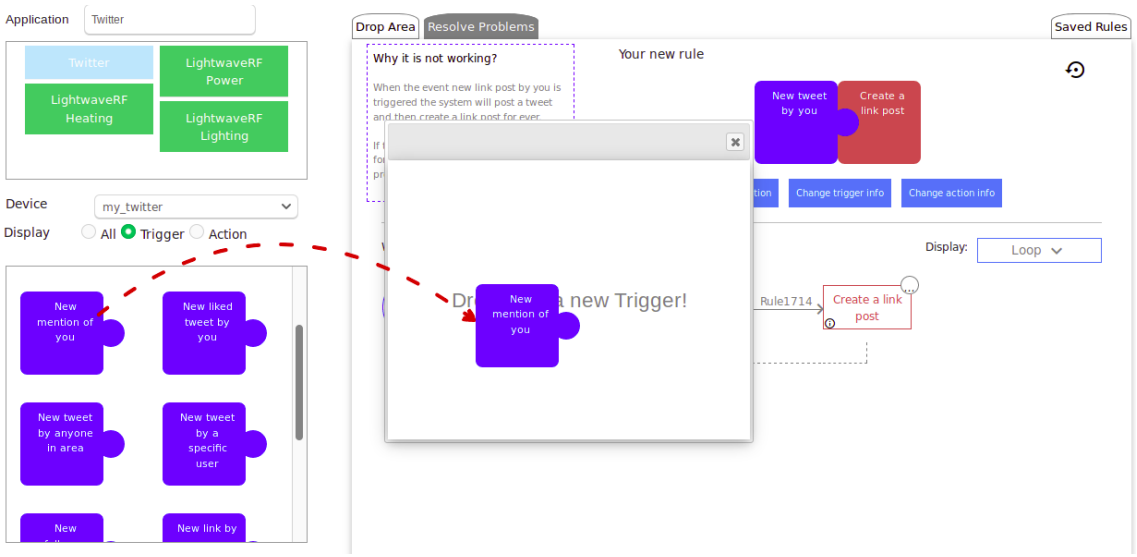


Figura 4.24: Modifica di un trigger o una action

Nel caso in cui invece l'utente decida di modificare solo i parametri del trigger o dell'action il sistema mostrerà una finestra che permetta di modificare i parametri inseriti precedentemente (figura 4.25). In questo caso il numero di modifiche concesse saranno infinite.

Dopo ogni modifica apportata alla regola, il tool andrà a ricontrollare, grazie all'ausilio del server, quali sono i problemi generati. Simultaneamente sarà aggiornata anche la rappresentazione nell'area di composizione.

Nel caso in cui la nuova regola non generi alcun problema l'area di risoluzione problemi sarà come mostrato in figura 4.26.

L'utente in ogni momento potrà decidere se salvare o eliminare la regola creata attraverso l'uso di due pulsanti situati sotto l'area di composizione e di risoluzione conflitti. Nel caso in cui la regola venga modificata nell'area di risoluzioni problemi questa sarà quella salvata dal sistema alla pressione del pulsante "Save" da parte dell'utente.

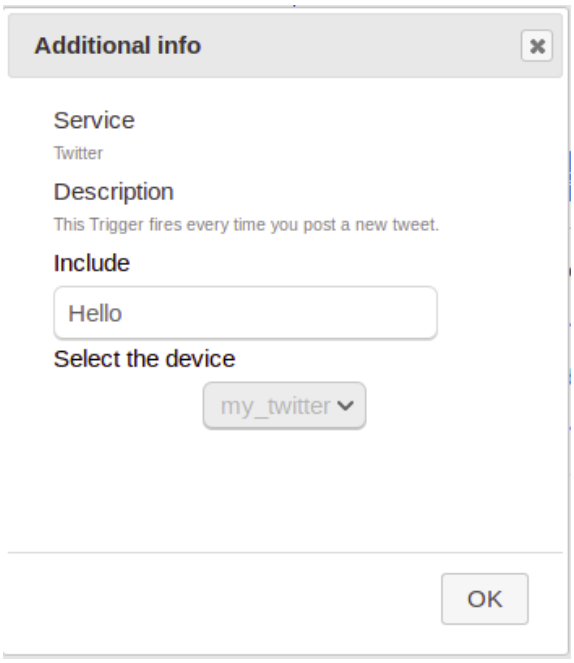


Figura 4.25: Modifica delle informazioni di un trigger o di una action

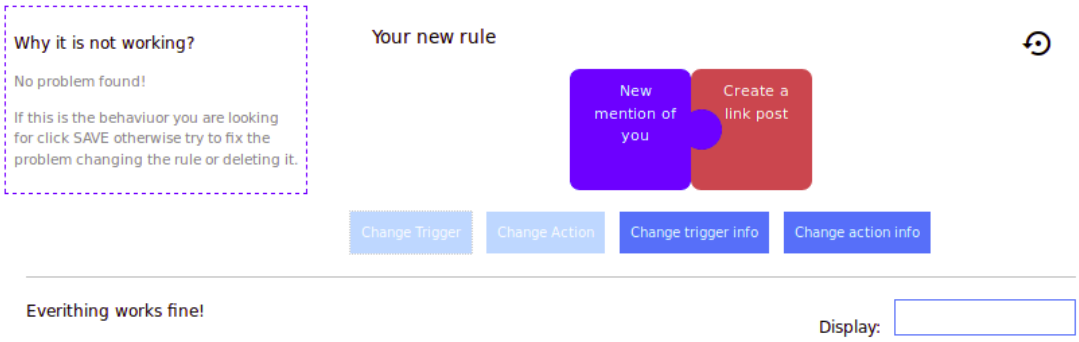


Figura 4.26: Schermata nel caso di risoluzione del problema

È possibile modificare la regola creata anche dalla rappresentazione mediante dataflow. L'action che compone la regola creata avrà infatti un pulsante che aprirà un menu che conterrà tutte le opzioni utilizzabili anche dall'area di modifica (figura 4.27).

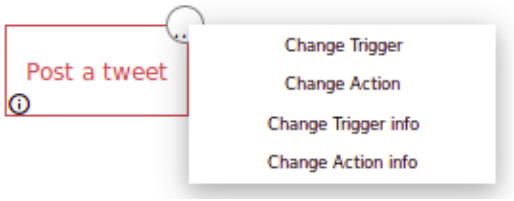


Figura 4.27: Modifica da dataflow

Anche questo caso il processo di modifica e le limitazioni a cui è soggetta saranno le stesse.

4.2.3 Regole salvate

In questa area, mostrata in figura 4.28, sarà possibile visualizzare la lista le regole salvate da un utente.

Questa area è accessibile in qualsiasi momento attraverso la selezione della tab corretta tra quelle mostrate in figura 4.16.

Per ogni regola, come mostrato in figura 4.29, sarà possibile visualizzare:

- il nome con cui la regola è stata memorizzata dal server
- una rappresentazione grafica della regola
- una rappresentazione testuale della regola
- un pulsante grazie al quale l'utente può eliminare la regola selezionata.

La rappresentazione visuale utilizzata è la stessa creata per il dataflow nell'area di risoluzione dei conflitti. Anche in questo caso su ogni componente della regola è presente un pulsante "i" per visualizzare più informazioni sul quel componente come il servizio da cui è fornito, il canale di utilizzo e informazioni aggiuntive.

La rappresentazione testuale è la classica rappresentazione di una regola che segue il paradigma trigger-action ed offre una sintesi testuale del funzionamento della regola.

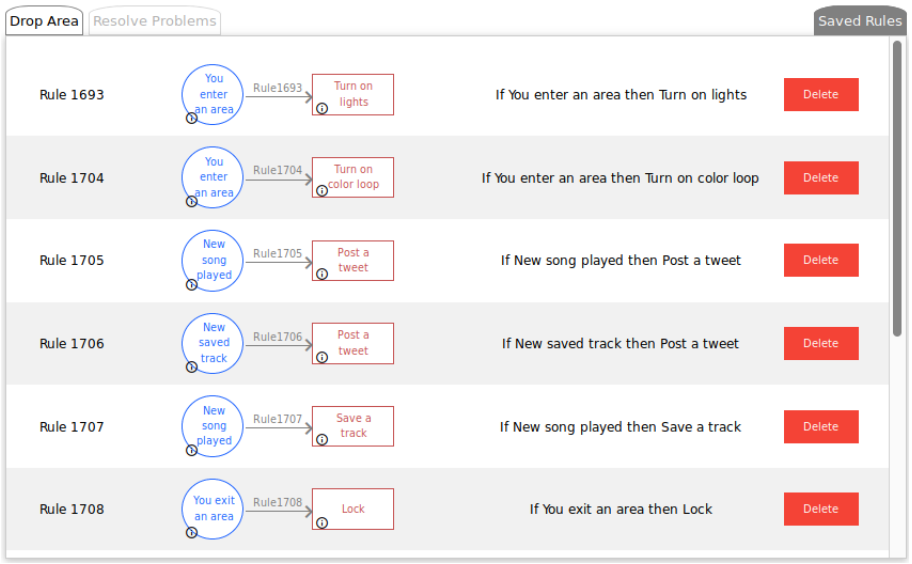


Figura 4.28: Lista delle regole salvate

Figura 4.29: Rappresentazione di una regola. (1) Nome della regola; (2) rappresentazione grafica; (3) rappresentazione testuale; (4) pulsante per eliminare la regola

Capitolo 5

Valutazione con utenti finali

Il tool EUD creato nel corso di questa tesi è stato valutato con 6 utenti finali. In questo capitolo sarà descritto il test effettuato e verranno presentati ed analizzati i risultati ottenuti.

5.1 Descrizione del processo di test

In questa sezione verrà descritto il processo seguito nella fase di test descrivendo le modalità, gli utenti che hanno preso parte a questo test, e le misure qualitative e quantitative che sono state raccolte.

5.1.1 Criteri di valutazione

L'obiettivo del test svolto è quello di valutare quali sono sono gli errori commessi più frequentemente da un utente finale (**QT1**), qual è il processo cognitivo seguito da un utente per risolvere i problemi (**QT2**), e quale linguaggio visuale risulta essere più intuitivo per la composizione e il debug di regole trigger-action (**QT3**).

Per raggiungere questo obiettivo il tool EUD sarà valutato in termini di:

- **Q1) Comprensibilità**. Gli utenti riescono a effettuare il debug di regole trigger-action?
  - **Q1.1)** Gli utenti riescono a comprendere il significato di loop, inconsistenza e ridondanza?
  - **Q1.2)** Gli utenti comprendono per quale motivo la regola definita può generare dei conflitti?
  - **Q1.3)** Gli utenti riescono a comprendere quali sono le modifiche da apportare alla regola composta per risolvere il conflitto generato?

- **Q2) Utilità.** Il tool EUD riesce ad aiutare un utente finale nella composizione di regole trigger-action?
  - **Q2.1)** Evidenziare i problemi generati è sufficiente a permettere agli utenti per l'identificazione dei conflitti?
  - **Q2.2)** Le spiegazioni testuali e grafiche sono utili per gli utenti per l'identificazione dei problemi?
  - **Q2.3)** La possibilità di modifica della regola è utile all'utente?
- **Q3) Usabilità.** Il tool EUD implementato è usabile?
  - **Q3.1)** Gli utenti riescono a comprendere come sono strutturate le varie aree e come interagire con loro?
  - **Q3.2**) Gli utenti riescono a comprendere come sono rappresentati graficamente e testualmente ridondanze, inconsistenze e loop?
  - **Q3.3)** Gli utenti riescono a capire come modificare una regola che ha generato dei conflitti?

5.1.2 Test

Il test consiste nella composizione di 12 regole e nel debug di quelle che generano dei problemi.

Inizialmente sarà chiesto ad ogni utente di rispondere ad un questionario iniziale dove dovrà valutare le proprie conoscenze informatiche e firmare un modulo per il consenso alla registrazione audio del test.

Successivamente verranno fornite delle informazioni contestuali su cosa sia un'ecosistema IoT e da quali dispositivi e servizi web può essere formato. Inoltre verranno fatti degli esempi pratici che fanno uso sia di dispositivi fisici che servizi virtuali.

Dopo aver concluso la parte introduttiva verrà spiegato come comporre una regola utilizzando una delle regole mostrate in Tabella 5.1, identificando esplicitamente il trigger e l'action con i relativi servizi da cui sono offerti e i canali di utilizzo.

Durante tutto il test, le regole saranno presentate ai partecipanti con la rappresentazione mostrata in Figura 5.1. In questa rappresentazione è possibile notare come il trigger sia associato alla parola chiave *If*, l'action alla parola chiave *Then* e al di sotto di entrambi siano elencati le informazioni addizionali specifiche. Tra le informazioni addizionali è possibile trovare sia il servizio da cui il trigger, o l'action, è offerto che il canale di utilizzo da selezionare.

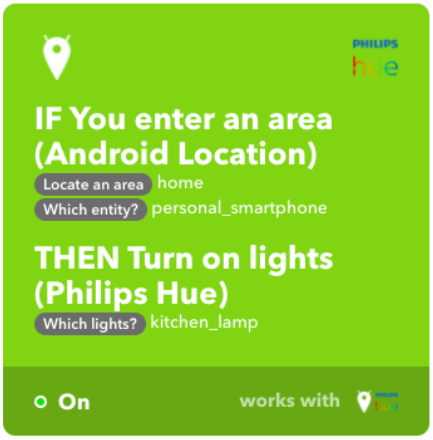


Figura 5.1: Rappresentazione della regola P1 fornita ai partecipanti al test

Per valutare se un utente senza alcuna nozione di informatica utilizzando il tool è in grado di comprendere i diversi tipi di possibili problemi è stato deciso di non spiegare ai partecipanti quali problemi possono generarsi dalla creazione di regole e di non introdurre i meccanismi di debug implementati dal tool.

Al termine della spiegazione iniziale non saranno fornite ulteriori informazioni sul funzionamento dell'interfaccia e l'utente potrà iniziare il test.

Il test consiste nella composizione di 12 regole, mostrate in Tabella 5.1, che in 5 casi genereranno dei problemi. In particolare saranno generate:

- 2 Ridondanze. La prima ridondanza è causata dalla composizione delle regole P1 e P3 (R1). La seconda ridondanza, causata dalla composizione delle regole P10, P11 e P12 (R2).
- 1 Loop generato in seguito alla composizione delle regole P4, P5 e P6 (L).
- 2 Inconsistenze. La prima inconsistenza è causata dalla composizione delle regole P1 e P2 (I1). La seconda ridondanza, causata dalla composizione delle regole P10, P11 e P12 (I2).

L'ordine di composizione delle regole è stato scelto a priori per permettere a tutti gli utenti di generare gli stessi problemi ma in ordine diverso. In Tabella 5.2 è possibile trovare l'ordine con cui ogni problema viene sottoposto ai partecipanti.

Dopo aver composto una regola, il tool comunicherà all'utente se sono stati rilevati problemi. Sarà poi l'utente a decidere se salvare o eliminare la regola composta. Se necessario l'utente potrà utilizzare l'area di "risoluzione di problemi" per avere più dettagli sui problemi generati e modificare la regola in modo opportuno per risolvere i conflitti.

Id	Servizio trigger	Trigger	Dettagli trigger	Servizio action	Action	Dettagli Action
P1	Android Location	You enter an area	**Locate an area**: home	Philips Hue	Turn on lights	**Which lights**: kitchen lamp
P2	Android Location	You enter an area	**Locate an area**: home	Philips Hue	Spegni le luci	**Which lights**: kitchen lamp
P3	Android Location	You enter an area	**Locate an area**: homea	Philips Hue	Attiva il color loop	**Which lights**: kitchen lamp
P4	iOS Photo	New photo added to album	**Album**: iosPhoto	DropBox	Add file from URL	**URL**: iosphoto **Folder**: drpb photos
P5	Dropbox	New file in your folder	**Folder**: drpb photos	Facebook	Upload a photo from URL	**URL**:drpb photo
P6	Facebook	New photo post by you	-	IosPhoto	Add photo to album	**URL**: facebook photo **Album**: iosPhoto
P7	Ios Location	You exit an area	**Locate an area**: work	SmartThings	Lock	**Which device**: office door
P8	SmartThings	Locked	**Which device**: office door	Homeboy	Arm camera	**Which camera**:office camera
P9	Homeboy	Camera armed	**Which camera**:office camera	SmartThings	Unlock	**Which device**: office door
P10	Amazon Alexa	New song played	-	Twitter	Post a tweet	**Tweet text**: I liked the Alexa song!
P11	Amazon Alexa	New song played	-	Spotify	Save a song	**Which track**: alexa song
P12	Spotify	New saved track	-	Twitter	Post a tweet	**Tweet text**:I liked the Spotify song!

Tabella 5.1: Le 12 regole composte dagli utenti durante il test

	Primo Problema	**Secondo Problema**	**Terzo Problema**	**Quarto Problema**	**Quinto Problema**
U1	I1	R1	L	I2	R2
U2	R2	I2	L	R1	I1
U3	R2	I1	R1	I2	L
U4	I1	R2	I2	L	R1
U5	I2	L	I1	R2	R1
U6	L	I2	R2	I1	R1

Tabella 5.2: Ordine di composizione di regole problematiche

Per ogni regola problematica composta sarà chiesto ad ogni partecipante di motivare le proprie scelte spiegando il problema segnalato sia nel caso in cui abbia deciso di salvare la regola che nel caso in cui abbia deciso di eliminarla.

Dopo aver composto tutte le regole sarà chiesto all'utente di compilare un questionario finale, in cui dovrà valutare il tool utilizzato, e gli sarà chiesto di fornire un commento su usabilità e funzionalità del tool testato.

5.1.3 Problemi generati

In questa sezione saranno analizzati i problemi che le regole composte durante il test possono generare e verranno illustrate alcune alcuna azioni che è possibile compiere per risolvere in modo corretto i problemi.

I1 Il problema I1, mostrato in Figura 5.2 è un inconsistenza generata in seguito alla composizione delle regole P1 e P2 (mostrate in Tabella 5.1).

In questo caso lo stesso trigger attiverà due azioni inconsistenti tra loro (accendi e spegni la lampadina). Per risolvere correttamente questo problema l'utente potrà scegliere di eliminare la regola definita o modificarla.

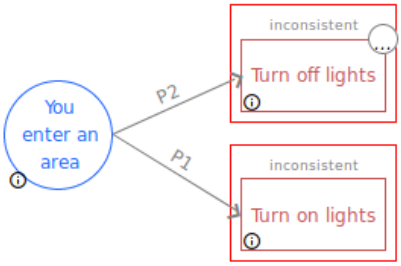


Figura 5.2: Rappresentazione grafica dell'inconsistenza I1 nell'area di risoluzione dei conflitti del tool.

R1 Il problema R1 è una ridondanza, mostrata in Figura 5.3, ed è generata in seguito alla composizione delle regole P1 e P3 mostrate in Tabella 5.1.

In questo caso lo stesso trigger attiverà due funzioni simili sulla stessa lampada (accendi - cambia il colore). Per risolvere questo problema saranno considerate corrette tutte le scelte, salvataggio o eliminazione della regola, se motivate coerentemente.

L Il loop L, mostrato in Figura 5.4, è generato in seguito alla composizione delle regole P4, P5 e P6.

In questo caso l'attivazione di una delle tre regole porta all'esecuzione infinita di questo loop. Questo perché la regola P4 attiva la regola P5, a regola P5 attiva la regola P6 e la regola P6 attiva la regola P4.

Questo problema può essere risolto solo eliminando una delle tre regole del ciclo o modificando il trigger, o l'action, di una delle tre regole.

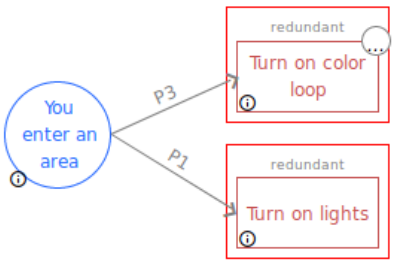


Figura 5.3: Rappresentazione grafica della ridondanza R1 nell'area di risoluzione dei conflitti del tool.

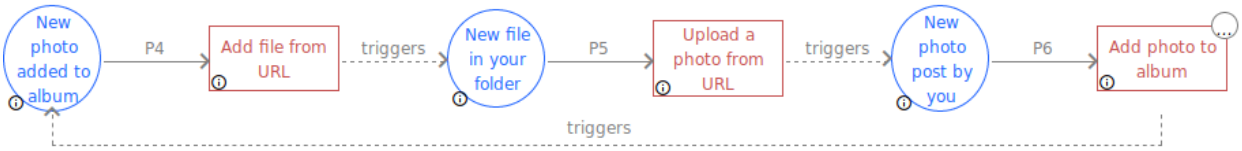


Figura 5.4: Rappresentazione grafica del loop L nell'area di risoluzione dei conflitti del tool.

I2 Il problema I2, mostrato in Figura 5.5, è un inconsistenza generata in seguito alla composizione delle regole P8, P9 e P10 (mostrate in Tabella 5.1).

In questo caso la composizione la regola P8 scatenerà la regola P9 che a sua volta scatenerà la regola P10. In questo caso l'inconsistenza sarà tra la regola P8 e P10 che non sono scatenate dallo stesso evento ma la cui esecuzione resta comunque legata.

Anche in questo caso le uniche due scelte corrette saranno l'eliminazione della regola P9 o la sua modifica.

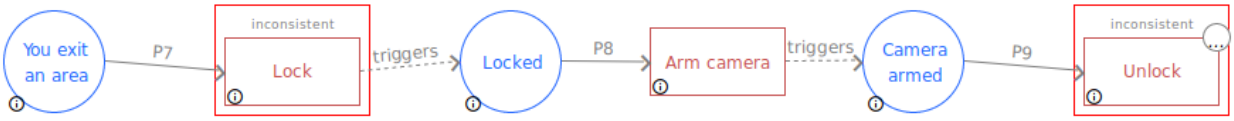


Figura 5.5: Rappresentazione grafica dell'inconsistenza I2 nell'area di risoluzione dei conflitti del tool.

R2 Il problema R2, mostrato in Figura 5.6, è una ridondanza causata dalla composizione delle regole P10, P11 e P12. In questo caso P10 e P11 saranno scatenate dallo stesso trigger, P11 attiverà P12 ma P10 e P12 eseguiranno la stessa azione.

Per risolvere in modo corretto questo problema l'utente dovrà eliminare la regola P12.

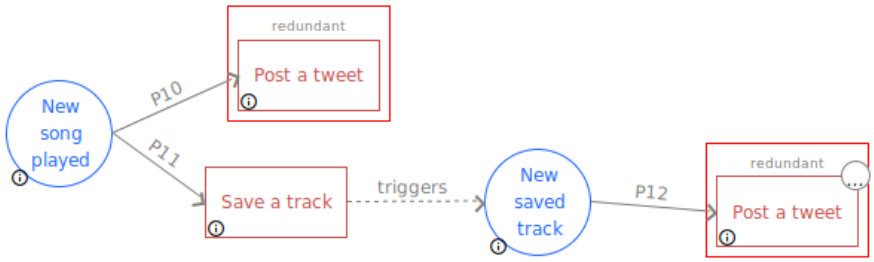


Figura 5.6: Rappresentazione grafica della ridondanza R2 nell'area di risoluzione dei conflitti del tool.

5.1.4 Partecipanti al test

Per partecipare al test sono stati selezionati 6 utenti: 3 uomini e 3 donne. Si è deciso di considerare come idonei, a partecipare allo studio, gli utenti senza alcuna conoscenza riguardante la programmazione. Prima di iniziare il test è stato chiesto a tutti i partecipanti di compilare un questionario iniziale per dichiarare il loro livello di tecnofilia, conoscenza tecnologica e capacità di programmazione .

I dati raccolti dai questionari iniziali sono mostrati in Tabella 5.3.

Da dati raccolti si può notare che:

- l'età media dei partecipanti è pari a 21.5 anni (SD 2.88);
- la tecnofilia, valutata secondo la scala Likert da 1 (Bassa) a 5 (Alta), raggiunge un punteggio medio pari a 3.5 (SD 0.54);
- la conoscenza tecnologica in generale, valutata secondo la scala Likert da 1 (Bassa) a 5 (Alta), ha media pari a 3.33 (SD 0.8);
- la dimestichezza con regole trigger-action, valutata secondo la scala Likert da 1 (Nessuna conoscenza) a 5 (Sono un'esperto), ha una media pari a 1 (SD 0);
- la capacità di programmazione, valutata secondo la scala Likert da 1 (Nessuna conoscenza) a 5 (Sono un'esperto), ha una media pari a 1.16 (SD 0.4);

Analizzando i dati presenti in Tabella 5.3 è possibile notare come due utenti abbiano dichiarato una capacità di programmazione pari a 2 ovvero di possedere delle conoscenze di programmazione ma non aver mai programmato. Nel primo caso l'utente U4 ha dichiarato di avere delle conoscenze rudimentali riguardante la programmazione PLC, nel secondo caso l'utente U5 ha dichiarato di possedere delle conoscenze basilari sulla programmazione con SAS. In entrambi i casi gli utenti sono comunque stati considerati idonei, per partecipare al test, perché entrambi hanno dichiarato di non avere alcuna nozione riguardante loop, inconsistenze e ridondanze.

Utente	Sesso	Età	Occupazione	Technophilia	Conoscenza tecnologica generale	Dimestichezza regole trigger-action	Capacità di programmazione
U1	Uomo	24	Studente di economia	3	3	1	1
U2	Donna	20	Studentessa di architettura	3	2	1	1
U3	Donna	19	Studentessa di architettura	4	4	1	1
U4	Uomo	21	Tecnico	3	4	1	1
U5	Donna	26	Analista	4	4	1	2
U6	Uomo	19	Studente di ingegneria fisica	4	3	1	2

Tabella 5.3: Dati raccolti dal questionario iniziale

5.1.5 Le misure

Per effettuare la valutazione del tool creato, sono state raccolte misurazioni qualitative e quantitative sia durante che al termine del processo di test.

Durante il test, attraverso l'uso di alcuni contatori, sono state raccolte le seguenti misure quantitative:

- numero di regole salvate
  - in presenza di un loop;
  - in presenza di una ridondanza;
  - in presenza di un'inconsistenza;
- numero di regole eliminate
  - in presenza di un loop;
  - in presenza di una ridondanza;
  - in presenza di un'inconsistenza;
- numero di regole modificate
  - salvate;
    - in presenza di un loop;
    - in presenza di una ridondanza;
    - in presenza di un'inconsistenza;
  - eliminate
    - in presenza di un loop;
    - in presenza di una ridondanza;

        * in presenza di un'inconsistenza;

- numero di regole ripristinate
  - in presenza di loop;
  - in presenza di ridondanze;
  - in presenza di inconsistenze;
- numero di accessi all'area di "Risolvi Conflitti"
  - in presenza di loop;
  - in presenza di inconsistenze;
  - in presenza di ridondanze;

Dopo ogni interazione con una regola generatrice di conflitti sono state raccolte le seguenti misure qualitative, attraverso delle domande orali:

- giustificazioni in caso di regole salvate;
- spiegazioni in caso di regola eliminata;

Al termine del test gli utenti hanno compilato un questionario finale attraverso il quale è stato possibile raccogliere le seguenti misure:

- quantitative
  - comprensibilità dei problemi;
  - comprensibilità del problema riferito alla singola regola;
  - comprensibilità della spiegazione del problema testuale;
  - comprensibilità della spiegazione del problema grafico;
- qualitative
  - utilità del tool;
  - comprensibilità del concetto di loop, ridondanza e inconsistenza;

5.2 Risultati del test

In questa sezione saranno presentati i risultati del test svolto. I risultati sono raggruppati in base alla scelta finale fatta dall'utente ovvero salvataggio, eliminazione o modifica della regola.

In generale tutti gli utenti hanno composto regole corrette e in 29 casi su 30 hanno utilizzato l'area di "Risoluzione dei conflitti" per analizzare i problemi comunicati dal tool.

5.2.1 Regole non modificate salvate

Su 30 regole segnalate in totale dal tool come problematiche ne sono state salvate, senza modificare alcun parametro, tre, ovvero il 10%, come mostrato in figura 5.7. Di queste tre regole due appartengono alla categoria di problemi non gravi, ovvero ridondanze, mentre una risulta essere un problema grave, ovvero loop.

Le ridondanze sono state salvate dagli utenti U1 e U3. In entrambi i casi i partecipanti hanno dimostrato di aver capito il problema fornendone una descrizione esatta: "*Accende la luce e ne cambia il colore ma per me non è un problema cambiare il colore della lampadina, quindi la salvo*" (U1), "*Accende e cambia il colore contemporaneamente secondo me è giusto così*" (U2). In questo caso gli utenti hanno interpretato correttamente sia la rappresentazione grafica del problema che quella testuale.

Nel caso del loop invece, l'utente U1 ha dimostrato di aver capito il problema segnalato dal tool ma, allo stesso tempo, l'uso di parole a lui non familiari come "URL" lo hanno portato a voler provare fisicamente il funzionamento delle regole composte per capire come risolvere il problema segnalato. L'utente U1 ha infatti dichiarato "*Ho capito il problema ma non so bene come risolverlo, quello che farei è salvare la regola e poi fare una foto con il telefono per capire quello che succede davvero*".

In questo caso l'utente U1 ha analizzato per molto tempo la rappresentazione grafica del problema senza però visualizzare mai i dettagli dei singoli componenti (trigger o action). Focalizzandosi su termini non importanti come "URL" invece di capire il processo generale.

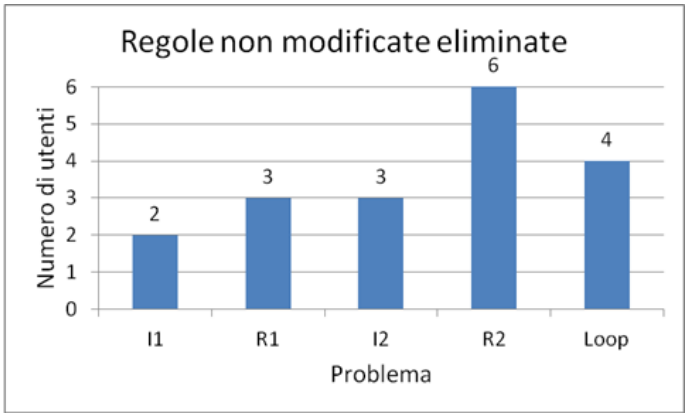


Figura 5.7: Regole non modificate salvate dagli utenti.

5.2.2 Regole non modificate eliminate

Su 30 regole gli utenti hanno deciso di eliminare, senza modificare alcun parametro, il 60% delle regole marcate come problematiche, ovvero 18 regole su 30.

In 17 casi su 18 gli utenti hanno dimostrato di aver capito il problema segnalato dall'interfaccia. In un solo caso l'utente U3 ha deciso di eliminare la regola R2 senza accedere all'area di "Risoluzione dei conflitti". Bisogna notare che questa era stata la prima regola composta dall'utente U3 e che nonostante non abbia capito il motivo del problema segnalato ha comunque deciso di eliminare la regola.

Le 18 regole eliminate, come mostrato dal grafico 5.8, sono composte da: 5 inconsistenze, 11 ridondanze e 4 loop.

Le inconsistenze sono risultate essere le regole eliminate più di rado. Questo è dovuto probabilmente alla facilità con cui gli utenti hanno capito come modificare in modo efficace le regole inconsistenti per risolvere i problemi generati. Sono state infatti eliminate solo nel 41% dei casi mentre le ridondanze sono state eliminate nel 75% dei casi e il loop nel 66% dei casi.

Gli utenti U3 e U4 hanno deciso di eliminare entrambe le inconsistenze. Nel caso dell'inconsistenza I1 l'utente U3 ha dimostrato di non aver capito appieno il problema. Dopo aver eliminato la regola ha infatti spiegato la scelta presa dicendo "*Se salvo queste regole il sistema non sa se deve accendere o spegnere le luci*". Questo fa capire come l'utente U3 non avesse capito che le regole sarebbero state eseguite quasi in contemporanea. Durante l'analisi di questo problema l'utente U3 si è concentrato principalmente sulla rappresentazione grafica del problema traendone una soluzione sbagliata. Nel caso della ridondanza R1, che rappresentava un grafico molto simile, l'utente U3 si è invece focalizzato sulla descrizione testuale e in quel caso ha compreso appieno il problema descritto, come analizzato nel paragrafo precedente. Al contrario nel caso dell'inconsistenza I2, l'utente U3, ha deciso di eliminare la regola avendo però compreso il problema generato.

L'utente U4 ha deciso di eliminare il 100% dei problemi rilevati. Per ogni problema ha saputo fornire una spiegazione corretta ed la sua analisi si è concentrata soprattutto sulla rappresentazione grafica del problema.

Le due regole eliminate più frequentemente sono state la regole legate ai problemi R2 e L. Il problema R2 è stato eliminato nel 100% dei casi e tutti i partecipanti hanno motivato questa scelta in modo corretto. La ridondanza R2, come spiegato nella sotto-sezione 5.1.3, è composta da una catena di regole (P11 e P12) che risultano essere un'estensione della regola P10 creando una ridondanza. In questo caso la scelta logicamente più corretta sarebbe stata quella di eliminare la regola P10 e salvare la regola P12. La nostra interfaccia non consentiva però di modificare

dall'area di "Risoluzione dei conflitti" la regola P10. Per questo motivo gli utenti hanno deciso di risolvere il problema eliminando la regola P12. Bisogna notare come a questo punto del test nessun utente ha avuto difficoltà a capire che il problema non fosse caratteristico di due sole regole ma che risiedesse nell'interazione di più regole.

Il problema L è stato eliminato nel 66% dei casi (4 volte su 6). Anche in questo caso la maggior parte degli utenti (3 casi su 4) è riuscito a comprendere il significato di loop e dell'esecuzione infinita delle regole. Sono riusciti inoltre a capire come l'eliminazione di una regola portasse alla risoluzione del problema. Solo l'utente U2 ha fornito una descrizione errata del loop dichiarando "*Io non voglio che faccia tutte queste cose in automatico*", però anche in questo caso va notato come l'utente avesse capito che le tre regole sarebbero state eseguite più volte di seguito.

In tutti i casi la spiegazione grafica è stata la rappresentazione del problema analizzata più frequentemente ma va notato come in alcuni, come ad esempio nel caso dell'utente U3, la spiegazione testuale abbia aiutato a capire che le azioni sarebbero state ripetute infinite volte.

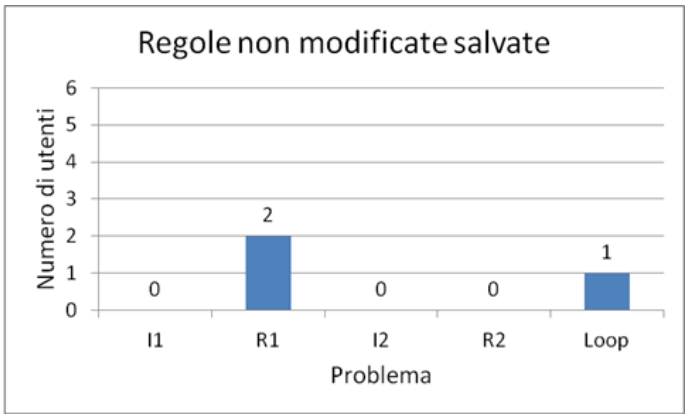


Figura 5.8: Regole non modificate eliminate dagli utenti.

5.2.3 Regole modificate

Il 66% degli utenti che ha preso parte a questo studio ha modificato in modo positivo almeno una regola. Sono state modificate il 30% delle regole composte e di queste ne sono state salvate il 78%, come mostrato in Figura 5.9.

Le regole che causavano inconsistenze sono risultate essere quelle modificate più frequentemente. L'inconsistenza I1 è stata modificata nel 66% dei casi mentre l'inconsistenza I2 è stata modificata nel 50% dei casi. In tutti i casi le modifiche apportate risultano essere logicamente e concettualmente corrette. Questo indica

che il problema era stato compreso nella sua totalità. Questo fenomeno è spiegabile perché le inconsistenze sono risultate essere gli errori non solo più facilmente compresi ma anche quelli più facilmente modificabili.

L'utente che ha utilizzato maggiormente l'area di modifica del tool è stato l'utente U6 che in totale ha modificato il 60% delle regole problematiche (3 regole su 5). Non tutte le modifiche hanno però risolto i problemi segnalati. Ad esempio nel caso del problema R1 dopo aver tentato varie combinazioni, tutte con esito negativo, ha deciso di eliminare comunque la regola composta. L'utente U6 è risultato essere l'unico a risolvere il problema del loop non eliminando la regola ma eliminando una delle connessioni tra le varie regole. Per fare ciò ha utilizzato un approccio "brute-force" provando varie combinazioni di coppie trigger-action per risolvere il problema. Durante queste prove ha utilizzato più volte la funzione di ripristino.

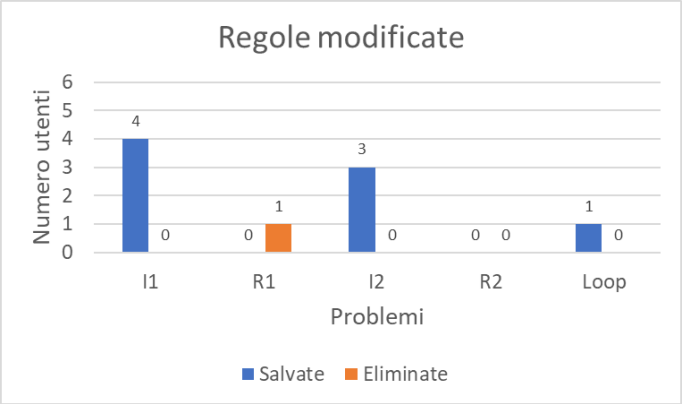


Figura 5.9: Regole modificate dagli utenti.

5.2.4 Questionario finale

Per il questionario di valutazione finale, compilato da ogni utente al termine del test, si è utilizzata la scala Likert da 1 (non comprensibile) a 5 (facilmente comprensibile).

Analizzando i dati raccolti, mostrati in Tabella 5.4, si ottiene che:

- il modo in cui i problemi sono evidenziati raggiunge una valutazione media pari a 4.5 (SD 0.54);
- i problemi evidenziati raggiungono una valutazione media pari a 4.16 (SD 0.75);
- la rappresentazione testuale utilizzato per descrivere il problema nell'area "Risolvi Conflitti" raggiunge una valutazione media pari a 4.5 (SD 0.81);

- la rappresentazione grafiche, ovvero il dataflow, utilizzato per descrivere il problema nell'area "Risolvi Conflitti" raggiunge una valutazione media pari a 4.5 (SD 0.83);

Analizzando i questionari dividendo gli utente per genere è possibile notare una distinzione tra uomo e donna soprattutto nelle valutazione della chiarezza delle rappresentazioni testuali e grafiche. Gli uomini hanno apprezzato la rappresentazione grafica molto più di quella testuale. In questo caso infatti la valutazione della rappresentazione grafica registra un voto medio pari a 4.66(SD 0.5), mentre quelle testuali hanno ottenuto una valutazione media pari a 4 (SD 1).

Le donne invece hanno apprezzato di più le rappresentazioni testuali rispetto a quelle grafiche. La rappresentazione testuale ha raggiunto una valutazione media pari a 4.66 (SD 0.58) mentre quella grafica ha raggiunto un punteggio medio pari a 4.33 (SD 1.15).

Utente	**I problemi evidenziati dal tool risultano essere**	**Il modo in cui i problemi sono evidenziati dal tool risultano essere**	**Le spiegazioni testuali del problema risultano essere**	**La rappresentazione grafica del problema risulta essere**
U1	4	4	3	5
U2	4	4	4	3
U3	5	5	5	5
U4	3	4	5	4
U5	5	5	5	5
U6	4	5	4	5

Tabella 5.4: Dati raccolti dal questionario finale; la scala di likert usata va da 1 (Non comprensibili) a 5 (facilmente comprensibili)

Al termine del test sono state raccolte anche delle valutazioni qualitative sull'usabilità del tool e sulle funzionalità che avrebbero voluto avere a disposizione per il debug delle regole.

Per quanto concerne l'usabilità del tool, tutti i partecipanti si sono dichiarati soddisfatti sia per l'area di composizione che per quella di risoluzione conflitti. L'utente U4 ad esempio ha dichiarato "*Dopo averci preso la mano è molto facile fare le regole e capire cosa non funziona*". In particolare, per quanto riguarda l'area di composizione tutti e 6 partecipanti sono stati soddisfatti sia della rappresentazione utilizzata per descrivere trigger ed action che del meccanismo di composizione della regola. Solo un utente, U1, ha suggerito che avrebbe apprezzato la possibilità di comporre più regole contemporaneamente.

Per quanto riguarda l'area di risoluzione conflitti molti partecipanti hanno espresso il desiderio di avere una spiegazione testuale più precisa, con servizi e canali di utilizzo, che evidenziasse in modo più netto le regole che risultano essere in conflitto.

Lo stesso desiderio di visualizzare più dettagli è stato espresso sugli elementi che vanno a comporre il dataflow.

Due utenti, U4 e U5, hanno anche espresso il desiderio di poter ricevere dei suggerimenti su come modificare una regola per risolvere il conflitto segnalato. Secondo il loro parere la presenza di suggerimenti come ad esempio "*Prova a sostituire il trigger X con il trigger Y*" può permettere ad utenti meno esperti di risolvere il problema e avere un punto di vista alternativo che gli consentisse di comprendere il problema.

L'utente U1 ha inoltre espresso il desiderio di poter visualizzare in aggiunta ai grafici e le rappresentazioni testuali anche delle animazioni che potessero simulare le regole rappresentate.

5.3 Discussione risultati

In questa sezione verranno discussi i risultati del test mostrati nella sezione 5.2. I risultati verranno analizzati valutando il tool su tre aspetti: comprensibilità, usabilità e utilità.

Comprensibilità Dai dati raccolti durante il test, sia quantitativi che qualitativi, emerge come in generale tutti gli utenti siano stati in grado di eseguire il debug di regole trigger-action (**Q1**). In particolare i partecipanti al test sono riusciti a comprendere i concetti di loop, inconsistenze e ridondanze (**Q1.1**).

Un dato importante è dato dalla facilità con cui gli utenti hanno compreso che i conflitti potessero essere causati dall'interazione di più regole (**Q1.2**). Al contrario di quanto riscontrato nello studio [12] nessun utente ha avuto problemi a focalizzarsi sul risolvere il problema segnalato nella sua complessità totale e non su quella della singola regola formata.

I partecipanti sono stati inoltre in grado di capire quali modifiche apportare alla regola per risolvere i problemi segnalati (**Q1.3**). Un dato importante è che anche persone caratterizzate da una bassa self-efficancy, come l'utente U2 che inizialmente ha più volte dichiarato "*No non lo so fare, non so niente di informatica non posso capirlo*", siano riuscite non solo a comprendere i problemi segnalati ma in alcuni casi a capire quali modifiche dovessero essere apportate, alla regola composta, per risolvere i problemi segnalati.

Inoltre è possibile notare come non si siano registrate delle differenze significative rispetto al sesso per quanto riguarda il numero di problemi risolti correttamente, contrariamente a quanto emerso in altri studi come ad esempio [10].

Utilità Il tool è risultato essere molto utile sia in fase di composizione che in fase di debug (**Q2**). In fase di composizione l'utilizzo della puzzle metaphor ha permesso

a tutti gli utenti di comporre facilmente le regole. La diversa colorazione dei pezzi del puzzle ha fornito, inoltre, un primo feedback ai partecipanti sui possibili conflitti che si sarebbero potuti generare. Ad esempio l'utente U3 quando ha selezionato per la seconda volta un pezzo, il cui colore risultava essere "usurato", ha dichiarato "*Ora creo di sicuro un problema*". In questo caso, l'utente U3, ha capito che l'utilizzo frequente dello stesso trigger può portare a dei problemi.

Le tecniche utilizzate per segnalare la generazione di problemi, sia in area di composizione che in area di "Risoluzione conflitti", si sono dimostrate efficaci (**Q2.1**).

La prima segnalazione del problema è fatta in area di composizione e la reazione tipica dei partecipanti è stata del tipo "*C'è un errore? In che senso*". Appena effettuato l'accesso all'area di risoluzione i partecipanti risultavano essere dubbiosi e in 4 casi su 6 la prima reazione è stata del tipo "*Non capisco il problema*". Successivamente, ad una seconda analisi delle varie descrizioni e rappresentazioni del problema, i partecipanti hanno progressivamente compreso il problema segnalato e sono stati in grado di prendere delle decisioni che fossero logicamente valide.

Questo ci fa capire come le rappresentazioni sia grafiche che testuali siano state di aiuto per tutti i partecipanti indipendentemente dalle loro conoscenze informatiche pregresse (**Q2.2**). Inoltre, la possibilità di modificare in modo semplice ed immediato una regola è stato un modo per gli utenti per essere sicuri di aver capito il problema (**Q2.3**).

Usabilità Dai dati raccolti il tool EUD creato si è dimostrato facilmente usabile da tutti i partecipanti (**Q3**) che sono riusciti ad interagire in modo efficace con tutte le aree presenti(**Q3.1**). La composizione mediante puzzle è risultata essere molto chiara ed intuiva, nessun partecipante ha avuto difficoltà nel distinguere i trigger dalle action e a comporre le regole. Il controllo dei conflitti effettuato durante la composizione ha permesso a tutti gli utenti di capire immediatamente se fosse stato rilevato o meno un problema. Il codice di colori utilizzato, verde, giallo e rosso, ha inoltre permesso agli utenti di capire l'entità del problema rilevato.

Analizzando i dati descritti nella sottosezione 5.2, è possibile affermare che la descrizione grafica unita alla descrizione testuale di un problema ha fornito ai partecipanti molte informazioni utili per la comprensione di loop, ridondanze e inconsistenze (**Q3.2**). Questo conferma che l'utilizzo di rappresentazioni differenti agevola gli utenti nel processo di analisi del problema come affermato nello studio [22].

L'utilizzo di una rappresentazione mediante dataflow per rappresentare l'interazione di regole che vanno a comporre un conflitto ha permesso ai partecipanti di capire le relazioni che intercorrevano tra le regole rappresentate. Le spiegazioni testuali hanno aiutato anche i partecipanti meno abituati all'uso di grafici a capire il

flusso di esecuzione delle regole rappresentate e di conseguenza identificare le cause che generavano i conflitti.

Un dato importante è dato dalla differenza di utilizzo delle due rappresentazioni, grafica e testuale, tra uomini e donne. Gli uomini si sono focalizzati, durante l'analisi del problema, maggiormente sulla rappresentazione grafica al contrario delle donne che hanno analizzato maggiormente la rappresentazione testuale. Il dato significativo è che per entrambi i sessi i risultati migliori si sono ottenuti quanto entrambe le aree erano analizzate in modo congiunto. Questo risultato conferma con quanto ipotizzato dallo studio [10].

Anche l'area di modifica di una regola è stata utilizzata positivamente da 5 partecipanti su 6 (**Q3.3**). In questo caso il controllo automatico fatto dopo ogni modifica ha permesso di evitare tutte le problematiche associate ad un "costo di esecuzione" alto e descritte nello studio [8].

Capitolo 6

Conclusioni

L'obiettivo di questa tesi è quello di realizzare un tool che possa aiutare un utente finale nel processo di debug di regole trigger-action per il controllo di dispositivi e sistemi IoT.

Il tool realizzato ha raggiunto ottimi risultati in termini di utilità e usabilità, e questo costituisce un primo passo verso la creazione di un'interfaccia EUD che possa aiutare ogni tipo di utente a gestire in modo facile e veloce i propri dispositivi e servizi IoT attraverso la creazione di regole trigger-action corrette.

Per raggiungere questo risultato sono stati compiuti diversi passi.

Il primo passo è stato quello di effettuare una ricerca in letteratura su tre temi principali: linguaggi visuali, processo di debug visto dal punto di un utente finale e metodologie di debug che utilizzano linguaggi visuali.

Alla fine di questa fase iniziale sono stati definiti:

- pro e contro per ogni tipo di linguaggio visuale (form-filling, block-programming e dataflow programming);
- i problemi più importanti che un utente affronta durante un processo di debug;
- il processo cognitivo seguito da un utente durante il debug;
- i paradigmi di debug che attraverso l'uso di linguaggi visuali riescono a fornire un aiuto ad un utente finale;

Terminata la ricerca in letteratura è stato fatto il secondo passo ovvero quello di design e progettazione di un tool per il debug di regole trigger-action. Per fare ciò sono state stabilite delle linee guida, da utilizzare nel proseguo della tesi, attraverso un'analisi preliminare. In questa analisi, partendo da quanto appreso dallo studio in letteratura, sono state definite delle metodologie che potessero aiutare un utente ad evitare di commettere determinati tipi di errori, ad esempio quelli sintattici, o in alternativa fornire delle informazioni che ne facilitassero il debug.

Al termine dell'analisi preliminare è iniziata la fase di design attraverso la creazione di 4 mockup.

A partire dai mockup generati, si è scelto di progettare ed implementare un sistema che assista gli utenti nella fase di composizione e debug delle regole.

Terminata la fase implementativa è stato fatto l'ultimo passo di questa tesi ovvero un test con utenti reali. Durante il test 6 utenti hanno provato a comporre delle regole e fare il debug di tutti i problemi che queste potevano causare. Il test ha avuto esiti molto positivi perché tutti gli utenti sono riusciti non solo ad interagire in modo efficace con il tool realizzato, ma sono anche riusciti a risolvere in modo corretto ogni tipo di problema generato nonostante non avessero alcuna conoscenza informatica. Questo fa capire come sia possibile creare dei tool EUD che permettano ad ogni tipo di utente di creare delle regole, in modo autonomo, che gestiscano un ecosistema IoT in modo sicuro e stabile, senza dover ricorrere all'aiuto esterno di tecnici specializzati.

6.1 Sviluppi futuri

Il mondo dei tool EUD per il debug sta ancora compiendo i primi passi e per questo sono ancora molti gli aspetti da migliorare. Il lavoro di questa tesi si presta, in particolare, a diversi sviluppi futuri che coinvolgono la composizione, la risoluzione dei problemi e lo sviluppo dell'interfaccia EUD.

Composizione L'area di composizione potrebbe essere migliorata permettendo agli utenti di creare più regole contemporaneamente e creare regole più complesse. Per regole più complesse si intendono le regole composte da più trigger e più action. Anche in questo caso l'uso del block-programming, con rappresentazione tramite puzzle, sarebbe funzionale perché continuerebbe a consentire agli utenti di creare regole complesse con meccanismi intuitivi.

Un ulteriore aiuto potrebbe essere quello di fornire una libreria delle regole composte più frequentemente, relative ad utenti con delle similitudine nella gestione dei propri dispositivi, fisici e virtuali.

Risoluzione dei conflitti Per quanto riguarda la risoluzione dei conflitti si potrebbero migliorare le descrizioni testuali dei problemi evidenziando meglio le azioni conflittuali. Inoltre la descrizione delle azioni potrebbe essere più precisa e con più dettagli su servizio da cui è offerta e sul canale su cui è trasmesso.

Un'ulteriore possibilità di sviluppo potrebbe essere quella di permettere agli utenti di poter modificare tutte le regole che contribuiscono alla creazione del problema. Inoltre, come proposto da alcuni partecipanti al test, sarebbe utile ricevere dal tool

dei suggerimenti su quali modifiche apportare alle regole per risolvere i problemi segnalati.

Sviluppo dell'interfaccia EUD Per quanto riguarda lo sviluppo dell'interfaccia di EUD, uno sviluppo certamente utile sarebbe quello di progettare anche un'area di simulazione simile a quella realizzata tra i mockup in fase di design, che permetta agli utenti di testare in modo tempestivo le regole composte indipendentemente dallo stato dell'ecosistema IoT.

Allo stesso modo potrebbe essere utile fornire agli utenti la possibilità di visualizzare, attraverso un linguaggio visuale congruo, tutte le regole eseguite in modo da poter analizzare tutte quelle situazioni in cui il sistema IoT ha dei comportamenti inattesi.

Bibliografia

[1] Ifttt, if this then that, www.ifttt.com.

[2] Massachusetts institute of technology , gameblox,gameblox.org/.

[3] Scratch,scratch.mit.edu.

[4] Louise Barkhuus and Anna Vallgårda. Smart home in your pocket. In *Adjunct Proceedings of UbiComp*, pages 165–166, 2003.

[5] A. F. Blackwell. First steps in programming: a rationale for attention investment models. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, pages 2–10, Sept 2002.

[6] Margaret Burnett, Curtis Cook, and Gregg Rothermel. End-user software engineering. *Commun. ACM*, 47(9):53–58, September 2004.

[7] J. Cao, S. D. Fleming, and M. Burnett. An exploration of design opportunities for "gardening" end-user programmers ideas. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 35–42, Sept 2011.

[8] J. Cao, I. Kwan, F. Bahmani, M. Burnett, S. D. Fleming, J. Jordahl, A. Horvath, and S. Yang. End-user programmers in trouble: Can the idea garden help them to help themselves? In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 151–158, Sept 2013.

[9] J. Cao, K. Rector, T. H. Park, S. D. Fleming, M. Burnett, and S. Wiedenbeck. A debugging perspective on end-user mashup programming. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 149–156, Sept 2010.

[10] Jill Cao, Yann Riche, Susan Wiedenbeck, Margaret Burnett, and Valentina Grigoreanu. End-user mashup programming: Through the design lens. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1009–1018, New York, NY, USA, 2010. ACM.

[11] Shi-Kuo Chang. Visual languages: A tutorial and survey. In Peter Gorny and Michael J. Tauber, editors, *Visualization in Programming*, pages 1–23, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[12] Luigi De Russis and Alberto Monge Roffarello. A debugging approach for trigger-action programming. In *Extended Abstracts of the 2018 CHI Conference*

on Human Factors in Computing Systems, CHI EA '18, pages LBW105:1–LBW105:6, New York, NY, USA, 2018. ACM.

[13] Arthur G Elser and Roger A Grice. Minimalism beyond the nurnberg funnel, 1998.

[14] Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Jill Cao, Kyle Rector, and Irwin Kwan. End-user debugging strategies: A sensemaking perspective. *ACM Trans. Comput.-Hum. Interact.*, 19(1):5:1–5:28, May 2012.

[15] C. Kelleher and R. Pausch. Lessons learned from designing a programming system to support middle school girls creating animated stories. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*, pages 165–172, Sept 2006.

[16] Cory Kissinger, Margaret Burnett, Simone Stumpf, Neeraja Subrahmaniyan, Laura Beckwith, Sherry Yang, and Mary Beth Rosson. Supporting end-user debugging: What do users want to know? In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '06, pages 135–142, New York, NY, USA, 2006. ACM.

[17] Andrew J. Ko and Brad A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM.

[18] Brad A. Myers, David A. Weitzman, Andrew J. Ko, and Duen H. Chau. Answering why and why not questions in user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 397–406, New York, NY, USA, 2006. ACM.

[19] M. Reisinger, J. Schrammel, and P. FrÃ¶hlich. Visual end-user programming in smart homes: Complexity and performance. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 331–332, Oct 2017.

[20] M. R. Reisinger, J. Schrammel, and P. FrÃ¶hlich. Visual languages for smart spaces: End-user programming between data-flow and form-filling. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 165–169, Oct 2017.

[21] A. Repenning and T. Sumner. Agentsheets: a medium for creating domain-oriented visual languages. *Computer*, 28(3):17–25, March 1995.

[22] N. Subrahmaniyan, C. Kissinger, K. Rector, D. Inman, J. Kaplan, L. Beckwith, and M. Burnett. Explaining debugging strategies to end-user programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, pages 127–136, Sept 2007.

[23] David T Tuma and Frederick Reif. *Problem solving and education: Issues in teaching and research*. Lawrence Erlbaum Associates, 1980.

[24] Jong-bum Woo and Youn-kyung Lim. User experience in do-it-yourself-style smart homes. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 779–790, New York, NY, USA, 2015. ACM.