UNIVERSITÀ DEGLI STUDI DI BRESCIA
FACOLTÀ DI INGEGNERIA
A.A. 1998-1999

ROBOTICA
Prof. Riccardo Cassinis

SURVEY: Linguaggi Visuali per la Navigazione Robotica

Allievi elettronici:

NEÉ LUCA
PAGANI DANIELE

1. Presentazione

La presente survey, sviluppata nell'ambito del corso di Robotica, si occupa di approfondire l'aspetto dell'applicazione di linguaggi visuali alla navigazione robotica.
L'applicazione di tecniche di programmazione visuale al problema del controllo dei robot ha recentemente avuto molta attenzione, grazie anche allo stimolo fornito da un'annuale competizione di programmazione di robot condotta al Visual Languages Conference.
L'intenzione del Visual Programming Challenge era di far convergere i vari approcci alla programmazione visuale, sia afferenti all'ambito della ricerca che a quello commerciale, su un problema specifico, con l'obiettivo di fornire un'analisi comparativa.
Questo lavoro si è concentrato sui risultati del Visual Programming Challenge, che hanno fornito notevoli spunti. Questo contesto, infatti, ha permesso di esaminare vari approcci al problema, con un immediato confronto tra le soluzioni sviluppate per risolvere i quesiti posti nella competizione.

La survey si articola nel seguente modo:
- Definizione del problema
- Introduzione al 1997 Visual Programming Challenge
- Concetti fondamentali
- Definizione dei linguaggi visuali
- Elenco dei linguaggi visuali per la robotica
- Confronto fra i linguaggi visuali per la robotica
- Caso di studio
- Bibliografia e documentazione
- Links

E' auspicabile che questo lavoro possa essere utile anche ad altri studenti, che si affacciano per la prima volta al mondo della robotica, dei linguaggi visuali e delle loro metodologie, quale sostanziale approfondimento sull'argomento.
Si ringraziano, oltre al prof. Cassinis che ha favorito la conoscenza e l'approfondimento di questo connubio tra robotica e linguaggi visuali, il prof. Mussio e il dott. Rizzi per l'aiuto nella ricerca del materiale e nell'impostazione del lavoro.

Neé Luca
Pagani Daniele

2. Indice

1. PRESENTAZIONE 2

2. INDICE 3

3. DEFINIZIONE DEL PROBLEMA 4

4. INTRODUZIONE AL 1997 VISUAL PROGRAMMING CHALLENGE 5

5. CONCETTI FONDAMENTALI 6

6. DEFINIZIONE DI LINGUAGGIO VISUALE 8

7. ELENCO DEI LINGUAGGI VISUALI PER LA ROBOTICA 11

Rule-Based 11
Dataflow 11
Controlflow 12
Equation-Based 12
Linguaggi di tipo Rule-Based 13
Cocoa 13
Altaira 19
RoadSurf 22
Linguaggi di tipo Dataflow 22
Prograph 22
Linguaggi di tipo Controlflow 25
VIPR 25
Create 33
Cwave 33
SeeDo 33
Linguaggi di tipo Equation-Based 35
Forms/3 36

8. CONFRONTO FRA I LINGUAGGI VISUALI PER LA ROBOTICA 36

9. CASO DI STUDIO 37

10. BIBLIOGRAFIA E DOCUMENTAZIONE 40

11. LINKS 43

3. Definizione del Problema

Da anni, la comunità di ricerca in programmazione visuale cerca il modo di applicare le tecnologie visuali per migliorare efficacemente il processo di programmazione. Gli approcci variano a seconda di come potrebbe essere ottenuto il miglioramento. Alcuni agevolano la comprensione dei modelli di programmazione convenzionali usando rappresentazioni visuali, altri usano modelli di programmazione alternativi che sfruttano aspetti della rappresentazione visuale.
Tema comune nella ricerca in programmazione visuale é l'aspetto visivo che, insieme ad altre forme di interazione tra uomo e computer, influenza la struttura del linguaggio e indirettamente determina "who is able to program what". Quindi le caratteristiche e i concetti di un linguaggio non solo riguardano "per che cosa" il linguaggio può essere usato, ma anche "chi" può usarlo.
Dalla scelta di appropriati concetti, caratteristiche e stili di rappresentazione, potrebbe essere possibile creare un linguaggio che abbia *ampia utilità* e *minimi ostacoli pedagogici*, che non sono necessariamente obiettivi cooperativi; in tal modo é concepibile un ampio numero di approcci. Da un lato linguaggi convenzionali come il C o il C++ hanno un'ampia utilità, ma sono abbastanza difficili da imparare; dall'altro, uno spreadsheet è usabile da molte persone che hanno una scarsa pratica di programmazione formale, ma gli spreadsheets hanno limitata applicabilità.[2]
Approfondiamo ulteriormente i concetti di *utilità* di un linguaggio di programmazione e di *minimi ostacoli pedagogici*.
L'*utilità* di un linguaggio è un indice della molteplicità del suo uso potenziale, cioè risponde alla domanda: qual è l'ampiezza dell'insieme dei problemi in cui il linguaggio è applicabile?
L'utilità di un linguaggio è funzione:
- dell'abilità ad accedere alle primitive necessarie, sia hardware che software;
- della capacità di astrazione e quindi di gestire la complessità;
- dell'eseguibilità dati tutti i supporti implementativi richiesti.

Quasi tutti i linguaggi possono, in linea di principio, fornire accesso ad un ampio insieme di primitive. Alcune volte, però, tali primitive sono fornite attraverso un secondo linguaggio che supporta l'implementazione e che è visibile almeno parzialmente.
La capacità di astrazione varia considerevolmente da linguaggio a linguaggio: l'astrazione procedurale è chiaramente presente in quasi tutti i linguaggi anche se è meno ovvia nei sistemi rule-based; l'astrazione sui dati può essere presente in almeno tre modi:
- nel linguaggio è offerta un'ampia varietà di astrazioni sui dati;
- il linguaggio supporta un meccanismo per definire nuove astrazioni sui dati;
- una o entrambe le precedenti due opzioni sono vere nel linguaggio che supporta l'implementazione.

Dal momento che i computer divengono sempre più potenti, l'insieme di problemi in cui l'esecuzione è critica diviene sempre più piccolo. Questo ha incrementato l'utilità di molti linguaggi.

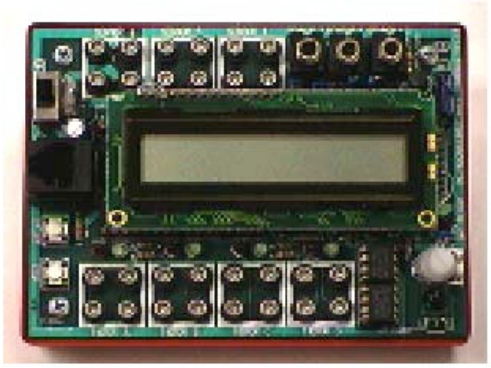
Per *ostacoli pedagogici* si intendono tutte quelle difficoltà che un utente incontra accostandosi a quel dato linguaggio. Si è già accennato che linguaggi fondamentalmente simili ai cosiddetti "linguaggi imperativi" (un esempio di tali linguaggi è il C), hanno generalmente significativi ostacoli pedagogici, mentre linguaggi specifici pensati per risolvere un particolare problema, spesso sono pedagogicamente semplici.[3]

Fatte queste premesse, il presente studio ha l'intenzione di approfondire l'aspetto dell'applicazione di linguaggi di programmazione visuale nell'ambito della navigazione robotica e di confrontare i vari approcci in termini di utilità, di ostacoli pedagogici e di altre caratteristiche significative in maniera tale da ottenere un'analisi comparativa. Partendo da questa base, un ulteriore obiettivo è quello di valutare come i linguaggi visuali presi in esame possono essere accostati all'implementazione di un particolare algoritmo di navigazione robotica.

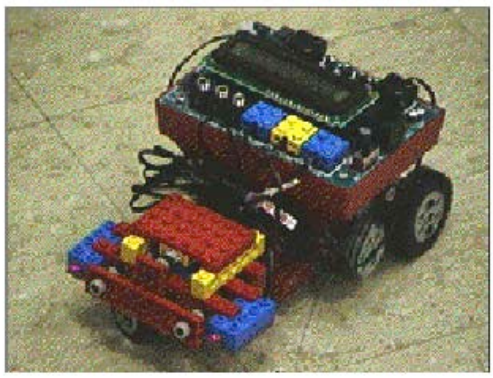
4. Introduzione al 1997 Visual Programming Challenge

Visto e considerato che gran parte del materiale utilizzato per realizzare questa survey orbita attorno al 1997 Visual Programming Challenge, vengono illustrati, con brevi cenni, i tratti salienti di questa sfida che è stata lo stimolo per l'interazione tra linguaggi visuali e robotica.

L'idea di applicare tecniche di programmazione visuale al controllo dei robot, ha avuto un considerevole interesse grazie al risultato del lavoro di Gindling [1] e altri, i quali hanno presentato un linguaggio visuale per controllare un semplice robot in un ambiente bidimensionale. Questa esperienza ha ispirato ad Ambler Allen la proposta di una competizione, da svolgersi al 1996 Symposium on Visual Languages, con lo scopo di usare linguaggi di programmazione visuale per controllare un robot in grado di svolgere un compito specificato. Il robot usato in questa competizione era il Programmable Brick[1], montato su un veicolo LEGO, che viaggiava su una pista LEGO, costruita con pezzi diritti e curvati, incroci a tre e quattro direzioni, tutto convenientemente codificato con del nastro nero per essere letto dai sensori ad infrarossi del veicolo.

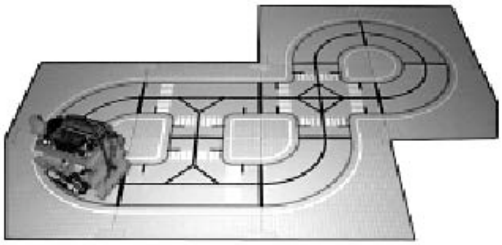


Programmable Brick



Veicolo LEGO

I compiti del robot erano quelli di esplorare e mappare circuiti di pista.
La competizione fu ripresa nel 1997 Symposium on Visual Languages. Il quesito della sfida fu esteso, richiedendo al robot, una volta mappata la pista, di cercare e percorrere il cammino più breve da un lato all'altro di un ostacolo posto sulla pista.[4]



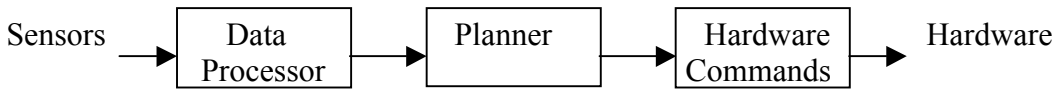
Situazione tipica

Tutti i dettagli relativi alla sfida possono essere trovati in [2] e [3].

[1] Il LEGO® Programmable Brick, sviluppato al MIT Media Lab, è un piccolo ma completo computer, che può essere usato congiuntamente con sensori ed attuatori per programmare il comportamento di oggetti meccanici.

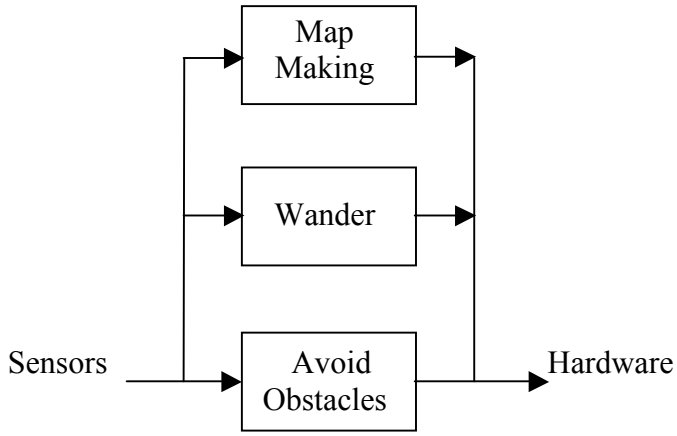
5. Concetti Fondamentali

I *robot* possono essere essenzialmente divisi in due categorie.
La prima categoria è rappresentata dai *robot industriali* usati nell'assemblaggio di prodotti o per analizzare campioni in un laboratorio. Tali robot hanno degli attuatori, ma sono spesso scarsamente equipaggiati di sensori. Eseguono sequenze di azioni ripetitive e abbastanza complesse, operano sempre in un ambiente completamente definito e non richiedono alcuna capacità di risolvere problemi. Sono normalmente programmati usando derivati di linguaggi di programmazione standard (per esempio RAPL-3, un derivato del Basic strutturato) utilizzando un approccio di tipo procedurale, combinati con una memorizzazione diretta di particolari movimenti.
La seconda categoria è rappresentata dai *robot autonomi,* i quali hanno sensori e attuatori ed operano in ambienti che sono parzialmente sconosciuti e possono cambiare. Tali robot percepiscono informazioni dal loro ambiente e calcolano valori per i loro attuatori, allo scopo di raggiungere determinati obiettivi. Un semplice esempio è la macchina LEGO usata nella sfida menzionata in precedenza; esempi più complessi sono il Mars Rover e veicoli autonomi sottomarini. I robot autonomi devono mostrare comportamenti reattivi uniti con capacità di risoluzione di problemi. Questo può essere ottenuto dalla semplice programmazione del feedback di sensori ed attuatori, usando un ordinario linguaggio di programmazione; comunque, un tale approccio potrebbe richiedere un programma di controllo per ogni nuova attività di risoluzione di problemi.[4]
In un robot mobile, un metodo di programmazione procedurale diventa difficile da utilizzare, poiché c'è una probabilità molto alta di raggiungere situazioni impreviste. A causa di ciò, una strategia di controllo reattiva, che utilizza regole per combinare gli input dei sensori con le informazioni di stato per produrre le uscite di controllo dei motori e i cambiamenti di stato, è un metodo comunemente usato con successo.[7]
Molte ricerche si sono dedicate ai sistemi risolutori di problemi per i robot con feedback.
Una delle prime applicazioni di logica per il controllo dei robot ha portato al linguaggio plan-representation STRIPS sviluppato per Shakey, il robot di Stanford. Più recentemente, forse incoraggiati dal progresso nell'esecuzione efficiente di programmi logici, sono state sviluppate logiche per trattare eventi real-time in generale e in particolare per il controllo dei robot.
L'affermato approccio di programmazione basato sulla logica per il controllo di robot mobili, cambiò nel 1986 in seguito *all'architettura subsumption di Brooks*, un modello di controllo che, attraverso una gerarchia di comportamenti indipendenti ma interattivi, ben si adatta a risolvere problemi. Sebbene l'architettura subsumption tratti molti dei più diretti aspetti del controllo, come evitare ostacoli e seguire percorsi, è meno utile per la pianificazione perché la gerarchia dei comportamenti diventa troppo complessa. Di conseguenza, sono state proposte molte architetture ibride che combinano controllo reattivo e pianificazione.[4]
Si amplia ulteriormente il discorso relativo al controllo dei robot in merito all'architettura subsumption di Brooks.
Il controllo classico di un robot è basato su una scomposizione sequenziale o orizzontale dei compiti di controllo in moduli funzionali. La figura seguente mostra questo approccio:



I sensori ottengono tutta l'informazione rilevante dall'ambiente che è trasformata in una forma conveniente per il *Planner* dal *Data Processor*. Il Planner usa questi dati per calcolare delle soluzioni per il problema di pianificazione, espresse in termini di azioni di alto livello. Queste azioni sono elaborate dal modulo Hardware Commands in istruzioni di basso livello per l'hardware del robot. Poiché ogni modulo richiede il suo input dal blocco funzionale alla sua sinistra, nella

situazione in cui qualche modulo fallisce, si determina il conseguente fallimento di tutto ciò che sta a destra.
L'architettura *subsumption*, anche conosciuta come *architettura behaviour*, usa una scomposizione verticale del compito in un insieme di comportamenti *task-achieving*, come mostrato in figura.



I comportamenti task-achieving sono sottosistemi indipendenti ed ognuno esegue un compito specifico; sono combinati per formare i livelli di competenza che forniscono al robot le sue capacità. A differenza dell'approccio orizzontale, ogni comportamento è indipendente, così il fallimento in qualsiasi livello non ha alcun effetto sugli altri.
La selezione del comportamento, in un sistema costruito in accordo con il modello dell'architettura subsumption, è eseguita da una gerarchia di controllo che deve garantire che, in ogni istante, solo un comportamento abbia il controllo del robot.[4] Si dettaglia meglio questo aspetto.
L'esecuzione di un comportamento è avviata da un cambiamento rilevato da un sensore. Molti comportamenti possono essere eseguiti in parallelo in risposta allo stesso cambiamento, ognuno specificando l'output che sarà messo in opera. I comportamenti hanno delle priorità: se più di un comportamento specifica un valore per un particolare input, il valore usato è quello dato dal comportamento con la più alta priorità.[5]

6. Definizione di Linguaggio Visuale

Nella letteratura, sono stati proposti molteplici formalismi per i linguaggi visuali. Quello riportato di seguito è stato scelto perché appare ben strutturato dal punto di vista logico ed inoltre offre la possibilità concreta del confronto diretto con l'autore.
Si procede con una serie di definizioni propedeutiche che portano a una definizione formale di linguaggio visuale.

Def.1 Un'*immagine digitale* è una stringa bidimensionale
$i : \{1,\ldots,r\} \times \{1,\ldots,c\} \rightarrow P$
dove r e c sono due integer e P è un generico alfabeto.

Si fa riferimento a un'immagine digitale chiamandola semplicemente immagine.

Def.2 Ogni elemento di un'immagine digitale è chiamato *pixel*.
Un pixel è descritto dalla tripla (r, c, p), dove $r \in \{1,\ldots,r_{max}\}$ è il pixel riga, $c \in \{1,\ldots c_{max}\}$ è il pixel colonna e $p \in P$.

Def.3 Una *struttura immagine* s è un sottoinsieme di un'immagine.
Una struttura immagine è specificata dalla sua funzione caratteristica:
$s\text{-}ch : \{1,\ldots,r_{max}\} \times \{1,\ldots,c_{max}\} \rightarrow \{0, 1\}$
e può essere definita come:
$s = \{(r, c, p) \in i \mid s\text{-}ch(r, c) = 1\}$
L'insieme di tutte le strutture immagine in un'immagine i è denotato da 2^i.

Def.4 Un *pictorial language* PL (linguaggio figurato) è un sottoinsieme dell'insieme I delle possibili immagini digitali:
$PL \subseteq I=\{i \mid \exists\, r, c \in \mathbb{N}, i:\{1,\ldots,r\} \times \{1,\ldots,c\} \rightarrow P\}$

La definizione di immagine come stringa bidimensionale (2-D) implica che i suoi pixel possono essere ordinati in diversi modi. Al contrario, una stringa unidimensionale (1-D) s (definita come $s: \{1,\ldots,n\} \rightarrow V$, dove V è un generico alfabeto) sfrutta solo l'ordinamento lineare e la conseguente definizione di concatenazione. L'ordinamento lineare non è privilegiato nell'analisi di stringhe 2-D, dove è invece importante specificare quale dei possibili ordinamenti è utilizzato e quindi quale definizione di concatenazione è adottata.
Vari tipi di immagine possono essere definiti in dipendenza dalla struttura algebrica che è imposta nell'alfabeto definito (per esempio immagini in bianco e nero, immagini a toni di grigio, ecc.).

Def.5 Una *semantica* di un'immagine i è una funzione
$sem: 2^i \rightarrow U$,
dove U è un insieme chiamato *universo di riferimento*.

Def.6 Un *modello caratteristico* è una coppia <s, u>, dove $s \in 2^i$, $u \in U$ e $u=sem(s)$.

Si noti che un osservatore specifica dei modelli caratteristici quando, guardando un'immagine, identifica in essa alcune strutture caratteristiche, le descrive (cioè fornisce una semantica per esse) e usa le coppie prodotte per interpretare l'immagine o ragionare su di essa. L'osservatore identifica tali strutture caratteristiche sulla base di obiettivi pratici nello specifico dominio di applicazione e di altre proprietà (geometriche, topologiche, ecc.).

Nell'interpretazione di un'immagine, gli elementi di U sono stringhe di simboli assegnati.

Def.7 Dato un alfabeto V e un insieme di insiemi di valori $\Omega=\{P_{a1},\ldots,P_{am}\}$, un *simbolo assegnato* su V è la (m+1)-upla costituita da un simbolo $v\in V$ e m proprietà $p_{ai} \in P_{ai,}\ 1 \leq i \leq m$.

Tutte le volte che ogni P_{ai} è un insieme finito di valori, si chiamerà $W=V\ X\ P_{a1}\ X\ldots X\ P_{am}$ l'*alfabeto dei simboli assegnati*.

Def.8 La *descrizione* d di un'immagine è una stringa di simboli assegnati su un alfabeto W
$d : \{1,\ldots,g\} \rightarrow W= V\ X\ P_{a1}\ X\ldots X\ P_{am}$

Def.9 Un *description language* DL (linguaggio di descrizione) è un sottoinsieme dell'insieme D delle possibili descrizioni:
$DL \subseteq D=\{d \mid \exists\ g\in\mathbb{N},\ d:\{1,\ldots,g\} \rightarrow W\}$

Si consideri DL come l'universo di riferimento.

Def.10 Un' *interpretazione* int di un immagine i è una semantica
$int : 2^i \rightarrow DL$.

La definizione sopra formalizza le funzioni calcolate empiricamente da un osservatore quando interpreta un'immagine. In verità, un immagine è descritta elencando i modelli caratteristici identificati in essa dall'interpretazione dell'osservatore.

Def.11 La *materialisation* mat di una descrizione di un'immagine d è una funzione
$mat : 2^d \rightarrow PL$,
dove 2^d indica l'insieme di tutte le possibili combinazioni di simboli assegnati ricavati da d.

Grazie a questa funzione si può sintetizzare un'immagine da una descizione data. Così una materialisation non è necessariamente una riproduzione dell'immagine originale.

Def.12 Una *visual sentence* vs (frase visuale) è una tripla <i, d,<int , mat>> dove i è un'immagine, d è una descrizione, int è una funzione di interpretazione e mat una funzione materialisation.

L'identificazione dei tre componenti permette di derivare programmi che:
a) analizzano una immagine data per ottenere un'interpretazione;
b) sintetizzano un'immagine da una descrizione data;
c) stabiliscono le relazioni tra una immagine data e una descrizione data;
d) gestiscono le visual sentences.

Def.13 Un *linguaggio visuale* è un insieme di visual sentenses.

Def.14 Una visual sentence <i, d, <int, mat>> è *faithful* (esatta) sse i=mat(d).

Def.15 Una visual sentence <i, d, <int, mat>> è *full* (piena) sse d=int(i).

Def.16 Una visual sentence <i, d, <int, mat>> è *complete* (completa) sse è full e faithful.

Def.17 Un *linguaggio visuale* VL è *complete* (*full o faithful*), se tutte le sue sentences sono complete (full o faithful).

I linguagi visuali sono informalmente definiti come linguaggi che usano costrutti visuali, cioè immagini di ogni tipo, come forme, diagrammi, icone, disegni, segni, ecc., per trasmettere un significato.
La nozione di visual sentence e linguaggio visuale data sopra soddisfa i progettisti che hanno bisogno di definire un sistema di programmi che:

a) analizzi una data immagine per ottenere un'interpretazione;
b) sintetizzi un'immagine da una descrizione data;
c) ragioni sulle immagini utilizzando le relazioni tra una immagine data e una descrizione data;
d) usi le visual sentences in un dialogo tra utente e computer.

Perciò, essi sono interessati a una definizione che identifica l'insieme di immagini e le descrizioni che sono l'oggetto e l'obiettivo deli analizzatori e dei sintetizzatori di immagini, così come all'insieme di funzioni che mettono in relazione modelli caratteristici nelle immagini con sottostringhe nelle descrizioni.[11]

7. Elenco dei Linguaggi Visuali per la Robotica

Sebbene la ricerca nell'applicazione di linguaggi visuali per la navigazione robotica è un'attività recente, diversi approcci si sono accostati a tale ambito, coinvolgendo linguaggi che possono essere classificati come *rule-based, dataflow, controlflow, equation-based.*
Ecco una breve panoramica delle quattro categorie sopracitate alla quale segue una presentazione dei singoli linguaggi specifici.

Rule-Based

Nell'approccio rule-based, il sistema ha uno stato corrente e un insieme di regole che trasformano il suo stato corrente in uno stato futuro. Ogni regola ha una precondizione e una specifica di trasformazione. Se la precondizione è soddisfatta, la trasformazione specificata è eseguita e il processo continua. Nella forma più semplice, assumiamo che tutte le precondizioni sono mutuamente esclusive o che se due regole sono entrambe applicabili, allora non ha importanza quale è usata. Con l'aumentare del numero di regole, in genere, questa assunzione semplificata non può sussistere e sono quindi richiesti altri meccanismi risolutivi più complessi.
Le soluzioni rule-based sembrano particolarmente efficaci per ridurre gli ostacoli pedagogici. Questo è, probabilmente, dovuto al fatto che il programmatore è sollevato da tutto ciò che riguarda il controllo, infatti può concentrarsi sulle situazioni e sulle loro soluzioni; il sistema si prende cura di gestire la valutazione della collezione di regole relative alle situazioni. Inoltre il programmatore affronta il problema cercando modelli e descrivendone le trasformazioni.
Sistemi rule-based sono particolarmente efficaci per esprimere il controllo di eventi reattivi; siccome il controllo di questi è spesso paragonabile a una macchina a stati, è facilmente codificato in semplici regole di trasformazione.
Negli approcci basati su regole possono esistere potenziali problemi dovuti al numero delle regole stesse, alla possibilità di conflitti fra loro e a come l'astrazione per dati e procedure è supportata senza complicare in modo eccessivo l'applicazione delle stesse. Inoltre, anche quando le regole singole sono facili da capire, ottenere una comprensione globale del programma può essere difficile.
Alcuni esempi di questi linguaggi sono: Altaira, Cocoa e RoadSurf.[2]

Dataflow

Un programma dataflow è una rete di computazioni, ogni nodo della quale riceve in ingresso un flusso di dati, esegue alcune computazioni su essi e ne restituisce i risultati. In un progetto dataflow puro, tutti gli aspetti relativi al flusso di controllo sono eliminati dalla considerazione dei programmatori. Il sistema gestisce il tutto determinando quando le singole computazioni sono pronte per essere eseguite, permettendo così al programmatore di concentrarsi sui rapporti tra i dati.
Gli eventi esterni sono rappresentati come flussi di ingresso alla computazione e il controllo esterno, come è richiesto ad esempio per il funzionamento dei motori del veicolo, è connesso al flusso di uscita dalla computazione.
Le soluzioni dataflow sono interessanti al fine della riduzione degli ostacoli pedagogici, poiché eliminano il bisogno di concentrarsi sul controllo.
Capire gli effetti dei cambiamenti di stato nel tempo può essere complesso.
I linguaggi dataflow tendono a supportare bene l'astrazione relativa alle procedure e ai dati.
D'altro canto, far fronte al controllo degli eventi reattivi è difficile nei linguaggi dataflow puri, perché gli eventi creano il bisogno di qualche controllo sull'ordinamento con cui tali eventi sono processati. Il modo in cui questo ordinamento è realizzato varia, ma inevitabilmente le soluzioni sono forzate a qualche compromesso, con l'accortezza che il compromesso raggiunto non debba aumentare la pratica richiesta per svolgere semplici compiti con il linguaggio.

Alcuni esempi di questi linguaggi sono: Prograph e Show and Tell.[2]

Controlflow

L'approccio controlflow è molto simile a quello dei convenzionali linguaggi di programmazione. La rappresentazione visuale è usata per esprimere tipici costrutti per il controllo del flusso computazionale come strutture condizionali, cicli e chiamate a procedure. Le rappresentazioni di solito impiegano sia diagrammi "box-and-arrow", così come sono usati dai diagrammi di flusso, che diagrammi annidati.
Le soluzioni controlflow sono particolarmente efficaci per risolvere problemi di controllo reattivo. D'altro canto, la maggior parte dei linguaggi controlflow usa una rappresentazione testuale convenzionale per manipolare dati. Questo limita le elevate potenzialità dovute alla migliore rappresentazione per strutture di controllo e per strutture di controllo alternative.
Una difficoltà è che i costrutti per il controllo del flusso computazionale del linguaggio convenzionale, non sono facilmente intuibili per un elevato numero di persone e richiedono notevole pratica. Quindi, le soluzioni controlflow sono andate oltre, trovando una rappresentazione alternativa per le strutture esistenti per il controllo del flusso.
Alcuni esempi di questi linguaggi sono: Create, Cwave, SeeDo e VIPR.[2]

Equation-Based

I linguaggi equation-based sono a volte chiamati form-based e/o funzionali. Essi sono simili ai dataflow puri in quanto si concentrano sulle relazioni tra i dati ed eliminano tutte le considerazioni sulle sequenze di valutazione. La differenza essenziale tra i linguaggi dataflow ed equation-based è che i linguaggi dataflow vedono i dati come un flusso continuo, mentre i linguaggi equation-based considerano i dati come valori discreti. Quindi, mentre i linguaggi dataflow considerano varie forme di feedback, i linguaggi equation-based usano una forma di ripetizione come la ricorsione.
Inoltre i linguaggi equation-based riducono gli ostacoli pedagogici attraverso l'eliminazione di concetti, come quelli che riguardano il flusso di controllo, e concentrandosi sulla manipolazione grafica dei dati.
Come per gli approcci dataflow, è difficile far fronte al controllo degli eventi reattivi perché gli questi creano il bisogno di controllo sull'ordinamento con cui sono processati.
Alcuni esempi di questi linguaggi sono: Form/3 e Formulate.[2]

Linguaggi di tipo Rule-Based

Cocoa

Introduzione.

Cocoa [15] (formalmente chiamato KidSim) è un tool di simulazione grafico per bambini. Questi creano dei "caratteri" su una "scena", disegnano il loro aspetto e forniscono "regole" per definire il loro comportamento.Le regole sono create mostrando cosa fa il carattere, usando la tecnica chiamata "programmazione tramite dimostrazione".
Un' idea intuitiva delle potenzialità del Cocoa può essere fornita attraverso questo elenco, in cui vengono puntualizzati i concetti fondamentali di questo linguaggio:

- *Rende visibile l'informazione dello stato: tutto ciò che è relativo all'operazione corrente è visibile sullo schermo.*
- *Massimizza l'operazione: c'è una relazione di causa-effetto tra le azioni dell'utente e la risposta del sistema.*
- *Permette di "creare attraverso le modifiche": permette alla gente di copiare e modificare le parti fatte, per crearne delle nuove, piuttosto che ricomporre tutto da zero.*
- *Supporta il "riconoscimento piuttosto che il richiamo": vedere e puntare è più facile che ricordare e digitare.*
- *Si concentra sulle cose concrete: la gente trova più facile manipolare istanze concrete piuttosto che astratte.*
- *Utilizza un modello concettuale familiare all'utente.*
- *Minimizza la distanza di traduzione: minimizza, cioè, la distanza tra la rappresentazione dei concetti nella mente dell'utente e la rappresentazione che il computer accetta.*

Particolare importanza ha la tecnica della "programmazione tramite dimostrazione", in quanto questo approccio consente anche ai bambini di programmare. Alan Key ha identificato il problema dell'inferenza nel come cambiare una immagine "Prima" nell'immagine "Dopo". Cocoa risolve questo problema facendo in modo che sia l'utente a dimostrare come si passi da "Prima" a "Dopo". Per questo motivo Cocoa è una combinazione di regole riscritte graficamente e della tecnica di "programmazione tramite dimostrazione". Alex Repenning ha sviluppato in contemporanea un sistema simile, che si chiama AgentSheets.

Cocoa è un ambiente di programmazione object-oriented, dove l'utente crea *caratteri*, li posiziona in una *scena* e fornisce loro delle *regole* che determinano il comportamento durante la simulazione.
Ogni carattere è un'istanza di una classe carattere che ha regole e variabili.
Tutte le istanze carattere ereditano lo stesso insieme di regole e variabili dalla loro classe, ma possono avere il proprio valore per le variabili.
Ogni classe carattere ha automaticamente associata una variabile chiamata "Appearance", che contiene la bitmap che viene usata per rappresentare il carattere sulla scena.
Per esempio, si consideri la simulazione di un giardino di fiori, ognuno dei quali è una istanza della classe carattere fiore.
Un fiore ha due aspetti (ossia due valori per la variabile appearance), uno mostra il boccio e l'altro mostra il fiore sbocciato. Un fiore ha anche una variabile chiamata "wetness", usata per tenere traccia di quanta pioggia sia caduta sopra il fiore e ha una regola che cambia l'aspetto da chiuso a sbocciato dopo che sia precipitata una certa quantità di pioggia.

Regole:

Le regole in Cocoa consistono di due immagini, "Prima" e "Dopo". Se Prima mostra lo stato attuale della scena, la regola contiene una lista di azioni che devono essere effettuate per convertire l'immagine Prima in Dopo.
Per esempio, la pioggia ha una regola (fig. c1) che fa sì che "se c'è un quadrato vuoto sotto di me, mi muovo in quel quadrato".

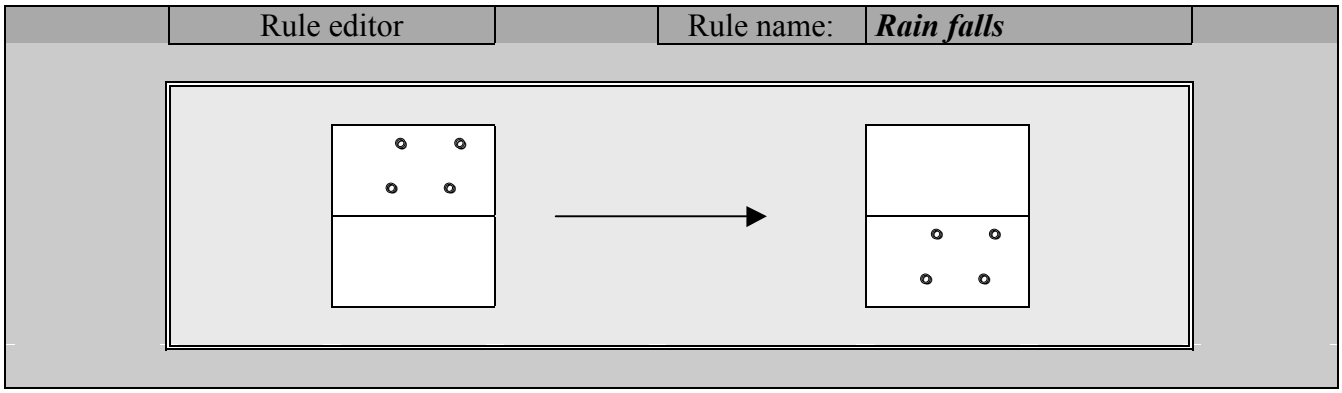


Fig. c1: un esempio di regola in Cocoa.

L'utente crea una regola mostrando l'azione, ossia disegnando la pioggia nel quadrato sottostante.

Esecuzione delle regole:

Cocoa usa un clock per controllare l'esecuzione. Ad ogni clock, ad ogni carattere sulla scena viene data la possibilità di eseguire una delle sue regole. Se ce ne è una, Cocoa esegue tutte le azioni associate a quella regola. I caratteri vengono esaminati in maniera sequenziale.

Variabili:

Cocoa potenzia le sue regole grafiche aumentando le loro variabili. Nei linguaggi puramente grafici, tutta l'informazione dello stato deve essere rappresentata graficamente. Permettendo l'utilizzo di variabili, ogni carattere può mantenere l'informazione sullo stato in modo non grafico e questa informazione può essere testata e modificata dalle regole. In più, Cocoa ha variabili globali, che possono essere usate da tutti i caratteri durante la simulazione.

Regole condizioni:

Oltre a confrontare il contenuto dei quadrati adiacenti con il "self"-carattere, all'immagine Prima di una regola si possono aggiungere test condizionali non grafici, che controllano i valori delle variabili. Per esempio, una regola per il fiore che sboccia, potrebbe richiedere che cadano almeno quattro gocce d'acqua sul fiore prima che questo sbocci.

Regole azioni:

Oltre alle azioni che muovono i caratteri da un quadrato all'altro, le regole possono contenere azioni che cambiano il valore della variabile.

Miglioramenti alle regole:

Oltre ai concetti visti, Cocoa ha una serie di miglioramenti che non cambiano le potenzialità espressive di Cocoa, ma piuttosto riducono il numero di regole necessarie per esprimere certe funzionalità.

- Quadrati "don't care": l'utente può marcare come "don't care" un quadrato nella figura Prima di una regola. Ciò significa che la regola può essere confrontata anche se il quadrato corrispondente sulla scena contiene oggetti addizionali che non appaiono nella definizione della regola.
- Jars (barattoli): il meccanismo dei jars in Cocoa permette all'utente di generalizzare una regola così che confronterà più istanze di una classe di caratteri. Per esempio, l'utente può creare un jar "pianta" e includere in esso le classi carattere fiori, erba e arbusti. Specificando che la classe fiore nella regola "piove sul fiore" si riferisce al jar pianta, questa regola sarà confrontata e incrementerà il valore della variabile wetness di ogni carattere fiore, erba e arbusti.

Insieme di regole:

Più regole possono essere raggruppate in un box chiamato RulSet. Normalmente le regole nel RulSet vengono testate in ordine dall'alto verso il basso. Questo ordine può essere diverso per tipi particolari di RulSet:

- *random RulSet*: alterna a caso le regole al suo interno prima che vengano testate;
- *do all and continue RulSet*: fa sì che tutte le regole al suo interno vengano eseguite e sospende l'esecuzione delle altre;
- *do in order RulSet*: contiene una lista di regole che sono eseguite in sequenza, una per ogni clock.

Pretest:

Cocoa consente che ogni RulSet abbia un pretest. Un pretest è solo la parte sinistra di una regola, ossia la figura Prima e le regole condizioni.

Il Challenge:

Analizziamo, ora, l'applicazione di Cocoa al Challenge.
Il Cocoa è stato testato su quattro diverse problematiche:

1. controllare un robot in tempo reale;
2. usare l'input dei sensori per identificare le diverse piastrelle della pista;
3. implementare un algoritmo per esplorare il tracciato;
4. implementare sul tracciato un algoritmo per trovare il cammino più corto.

Ricordiamo, inoltre, che i linguaggi visuali sono stati valutati in base ai seguenti criteri:

- completezza della soluzione;
- ampio numero di utenti;
- soluzione più piacevole da implementare per un inesperto;
- migliore presentazione.

Soluzione in Cocoa:

Innanzitutto Cocoa doveva essere esteso per permettere il controllo della macchina attraverso una cavo seriale. Così c'era necessità di una interfaccia; c'erano due soluzioni plausibili: la prima era quella di introdurre variabili "volatili", sincronizzate con i valori reali della macchina Lego e che

potevano essere letti (tramite i sensori) e scritti (per i movimenti dei motori) come semplici variabili globali; l'altra possibilità era di estendere le regole di Cocoa con regole test e regole azione, utilizzando una interfaccia suggerita da Alex Repenning.
Le regole test introdotte riguardavano i sensori della macchina Lego, che potevano essere letti per un specifico insieme di valori (per esempio, "sensore 3 riconosce il nero, sensore 4 riconosce il bianco"..); le regole azione ammettevano una speciale azione per settare le velocità dei motori. E' stato scelto questo secondo approccio, per rendere più "leggera" la parte di interfaccia a più basso livello.

Una regola semplice:

La regola rappresentata in figura c2 controlla se la macchina è su un pezzo del tracciato che è già stato esplorato. Se è vero, mette un "segui freccia" nella variabile "activity". Questa regola mostra due caratteristiche avanzate in Cocoa- i quadrati "jar" e "don't care"- già presentati in precedenza.

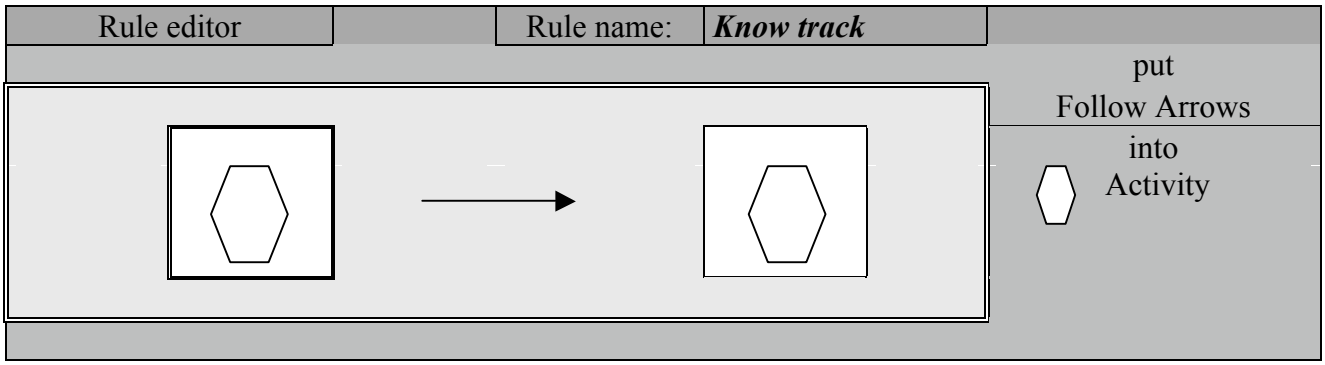


Fig. c2: un esempio di regola in Cocoa.

Al Visual Programming Challenge c'erano 4 diversi tipi di pezzi di tracciato: dritto, a curve, incrocio a 3 strade, incrocio a 4 strade. Poiché era difficile distinguere subito tra un incrocio a 3 strade e un incrocio a 4 strade, è stato introdotto un quinto tipo chiamato "incrocio a 3-4 strade".
La regola in figura c2 è applicabile a ogni pezzo della traccia escluso l'incrocio a 3-4. Perciò è stato usato un jar chiamato "non 3-4", che contiene, cioè, tutti i tipi di traccia non con incroci a 3-4 strade.
Un quadrato (fig c3) poteva essere ispezionato clickando con il pulsante del mouse; clickando sul jar si poteva vedere quali oggetti c'erano in esso (fig c4).

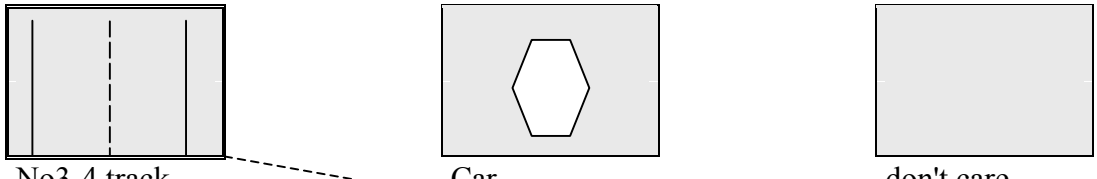


Figura c3: quadrati presenti sulla scena.

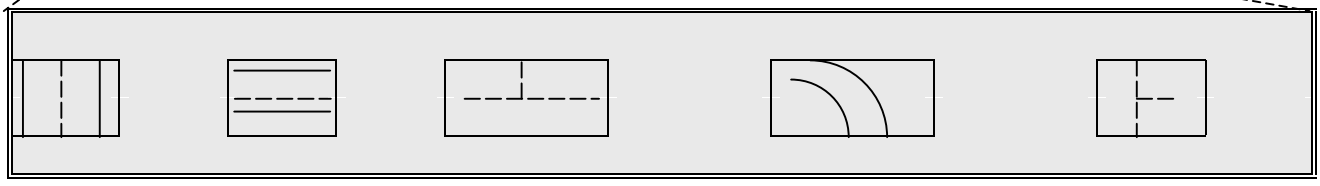


Figura c4: Jar

Controllo del robot in tempo reale:

Cocoa è un sistema basato su regole e perciò è abbastanza semplice implementare un algoritmo base per mantenere la macchina sulla linea centrale. C'è bisogno di 3 regole:

1. se la macchina è esattamente sulla linea, allora vai dritto,
2. se la macchina è un pò a destra, correggi la sua corsa a sinistra;
3. se la macchina è un pò a sinistra, correggi la sua corsa a destra.

La parte sinistra della regola ha il test dei sensori e la parte destra ha la azione corrispondente alla situazione rilevata. Fig. c5.

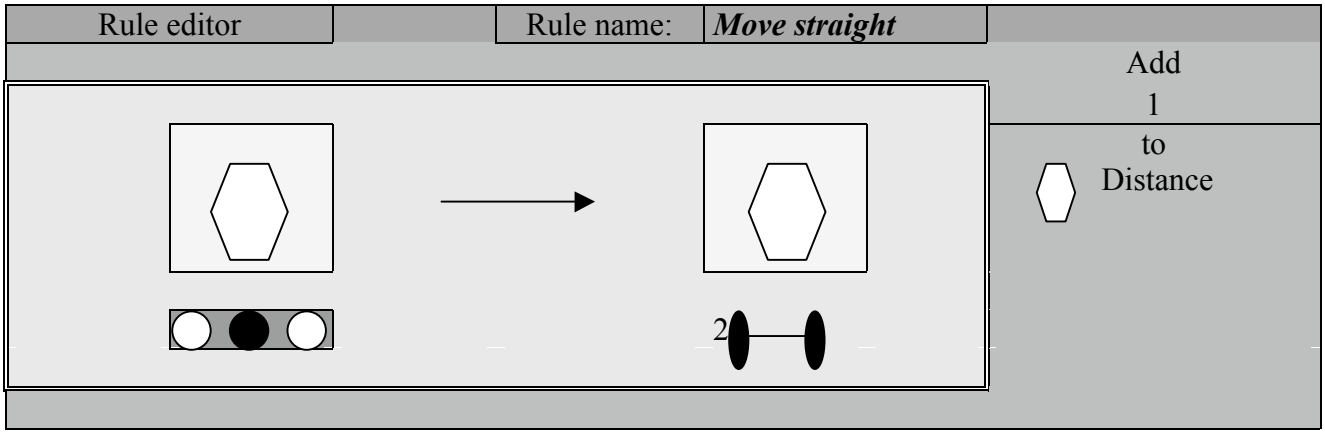


Figura c5: una regola in Cocoa per seguire la linea centrale.

Utilizzo dei sensori per identificare diverse piastrelle stradali:

Il problema è riconoscere su quale piastrella la macchina si trovi. Ovviamente questo si traduce nel distinguere tra un pezzo con intersezioni o senza intersezioni, e quindi tra tre tipi di incroci (3 sinistra/destra, 4 o 3 centrale) e tre tipi di non intersezioni (gira sinistra, gira a destra o vai dritto).

Algoritmo per esplorare il tracciato:

Il carattere macchina deve ora guidare il suo "cugino fisico" sopra ogni pezzo del tracciato, facendo seguire la linea centrale e uscire dalla piastrella. Può anche distinguere tra non intersezione e incrocio a 3 strade sinistra/destra. Non può, tuttavia, distinguere tra un incrocio a 3 vie venendo dal centro e un incrocio a 4 vie, perché sono uguali dal punto di vista della macchina. La macchina marca questi per una esplorazione successiva e disegna sulla mappa una piastrella "3 o 4". In Cocoa, la mappa consiste in caratteri "piastrella stradale" senza regole.

Teoria contro Pratica:

In pratica, tuttavia, la macchina è troppo lenta e le batterie troppo deboli per permettere l'applicazione dei principi visti. Così è stato implementato un algoritmo per trovare il prossimo pezzo di tracciato non esplorato: ogni volta che la macchina passa su una piastrella già esplorata, manda un carattere "macchina scout" nella simulazione. La macchina scout trova la prossima piastrella da esplorare, lasciando una freccia che la macchina reale dovrà seguire. La macchina scout effettua una ricerca in profondità del tracciato. Le figure c6 e c7 mostrano le regole per la macchina scout e possono essere lette come: "se la macchina scout è su una piastrella che ha uno sbocco a est AND c'è una piastrella non visitata a est, allora sparisci e lascia una freccia che punta a est".

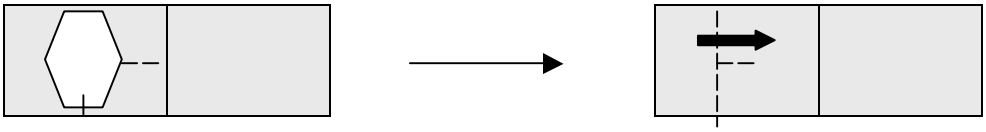


Figura c6: la regola "trova la piastrella inesplorata" per lo scout

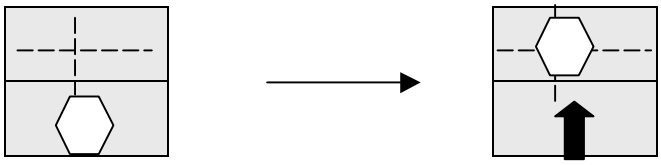


Figura c7: la regola del movimento per lo scout.

Trovare l'incrocio 3 o 4:

Rimane il problema di riconoscere tra un incrocio a 3 o a 4 strade. La strategia è di far sì che la piastrella "3 o 4" sia esplorata da diversi punti di vista; solo allora si può stabilire con certezza il tipo di incrocio. Per far ciò si è creato un nuovo carattere macchina scout, dopo che la prima macchina scout ha determinato che non ci sono più piastrelle inesplorate sul tracciato. Per crearle si è copiata la prima macchina scout e la si è chiamata "3 o 4 scout". Si sono rimosse le 4 regole e si sono introdotte 4 nuove regole. Tutte le regole per la ricerca in profondità del tracciato rimangono le stesse.

L'algoritmo completo per la macchina:

Il carattere macchina ha una variabile "activity" e 4 insiemi di regole che rappresentano gli stati: "guarda per un marker di intersezione", "segui il tracciato", "segui le frecce", "muoviti verso la prossima traccia".
Questo insieme di regole hanno i pretest per fare un controllo su "activity", così solo un insieme di regole viene visitato in un clock. Il carattere macchina ha anche altre variabili: la distanza stimata da una data piastrella, una direzione assoluta nella forma di 4 differenti aspetti, e una direzione relativa (per esempio "sinistra") per la navigazione nelle intersezioni.

- " guarda per un marker di intersezione" contiene regole per seguire la linea centrale e per riconoscere i markers di intersezione. Contiene anche regole-contatori che contano il numero totale di step (piccoli movimenti in avanti) e il numero di aggiustamenti durante il percorso (correzione a sinistra o a destra). La macchina lascia questa attività dopo un certo numero di step o dopo aver riconosciuto uno o due markers di intersezione. Quando ne trova uno, deposita la giusta piastrella di intersezione.
- "segui il tracciato" semplicemente fa seguire il percorso fino alla fine della piastrella. Se la piastrella non è conosciuta, deposita una piastrella direzionale (dritto, sinistra, destra). La macchina deposita una piastrella "vai dritto" se il numero di correzioni a sinistra e a destra è uguale ad una certa soglia, se no curva. Il tipo di curva è dedotto dalla direzione assoluta e da quella relativa.
- "segui la freccia" consente di seguire le frecce sopra una piastrella intersezione; se non ci sono frecce, crea un carattere "scout" che le mette. La freccia punta nella direzione dove la macchina si muoverà per trovare la prossima piastrella non visitata.
- "muovi verso la prossima traccia" è necessario per navigare in modo sicuro verso la prossima piastrella. Il carattere macchina è in questo stato quando la macchina fisica è sulla striscia nera tra due piastrelle.

Questa è stata la soluzione del problema obbligatorio. La richiesta successiva era la seguente: dopo che il tracciato è stato interamente mappato, qualcuno mette un marker-destinazione sullo schermo e la macchina reale deve arrivarci; per risolverlo sono state semplicemente introdotte 4 regole per la macchina scout per individuare la destinazione.

Final Challenge: l'algoritmo del cammino più breve in Cocoa:

Il confronto successivo, presentato due settimane prima della conferenza, richiedeva di implementare un algoritmo per determinare il cammino più breve verso la destinazione. Per far questo è stata aggiunta una regola e una variabile: la variabile contiene la lunghezza del percorso più corto dalla piastrella attuale a quella dove vogliamo andare. La regola dice: "se una piastrella vicina ha la lunghezza più bassa della mia lunghezza più 1, allora setto la mia lunghezza alla sua più 1. Ovviamente questa regola va scritta per ognuna delle 4 direzioni.

Propagazione della lunghezza del cammino più corto:

Quando la macchina ha mappato l'intera configurazione, la variabile globale "activity" viene settata a "trovato mattone" e la macchina manda un'altra versione dello scout "trova mattone scout". Ad un certo punto la macchina trova l'ostacolo e la variabile activity viene settata a " propaga la lunghezza" e la variabile "lunghezza" nella piastrella sul lato del mattone viene settata a zero. La lunghezza viene propagata attraverso la configurazione e la macchina aspetta e controlla se il contatore globale è più grande di uno e accresce questo contatore. Se la piastrella cambia il suo valore, setta anche a zero il contatore. Tuttavia, se passano due cicli e nessuna piastrella è adatta, il contatore globale è settato a 2- condizione di uscita per l'attività "propaga lunghezza". La macchina, quindi, segue le piastrelle con il percorso più corto per arrivare alla destinazione.

Conclusione:

Cocoa era stato testato su oltre 300 bambini di età compresa tra i 5 e i 15 anni. Tutti hanno creato caratteri e regole dopo 5 minuti di introduzione. Cocoa non è progettato per essere un linguaggio di programmazione generale, ma è un ottimo compromesso tra facilità di uso e potenza espressiva del linguaggio. Con il Visual Programming Challenge si è dimostrato che con Cocoa:

1. è possibile implementare algoritmi complessi;
2. si possono controllare robots.

La soluzione al Visual Programming Challenge ha richiesto 130 regole (molto delle quali simmetriche per le 4 direzioni possibili); ma il punto di forza di questo linguaggio è da ricercarsi soprattutto nella facilità di sviluppo e di programmazione.

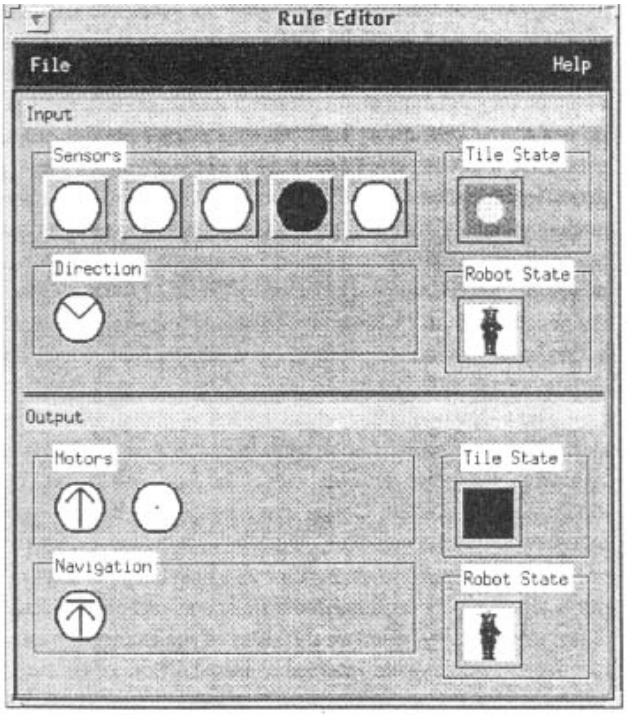
Altaira

Introduzione.
Altaira è un linguaggio visuale progettato per il controllo di piccoli robot mobili (per esempio il robot LEGO). E' un linguaggio rule-based che combina le informazioni fornite dai sensori, la navigazione e lo stato corrente, per determinare le azioni che il robot deve compiere e i cambiamenti opportuni nello stato del sistema.
Il linguaggio è abbastanza semplice da essere usato da bambini per sviluppare semplici comportamenti per il robot, ma nello stesso tempo è anche abbastanza potente da essere impiegato per risolvere i problemi del 1997 Visual Programming Challenge così come per implementare un algoritmo dimostrativo della Macchina di Touring.
Il linguaggio è innovativo per quanto riguarda:

- *la rappresentazione specifica e separata dello stato globale e locale*
- *il trattamento uniforme per lo stato e i sensori di input e per lo stato e gli attuatori di output.*

Altaira può essere rivisto come l'intersezione di due aree di ricerca: controllo reattivo dei robot e linguaggi visuali rule-based.
Anche se originariamente pensati come un modello dei processi umani per risolvere problemi, i sistemi rule-based sono stati usati con successo nell'implementazione di strategie di controllo reattivo per i robot mobili (cioè, strategie di controllo in cui il robot esibisce comportamenti come reazione a degli eventi). L'uso di un sistema basato su regole per implementare una strategia di controllo reattivo è stato proposto, ad esempio, nella programmazione del Mars Rover della NASA.
Una particolare forma di controllo reattivo di robot che usa una gerarchia di regole è l'architettura subsumption di Brooks. In un'architettura subsumption, alle regole sono assegnate delle priorità: le regole con più alto livello di priorità sono in grado di escludere quelle con priorità inferiore, che sarebbero altrimenti permesse (cfr. pg 8).

Altaira è un linguaggio visuale pensato per programmare robot mobili in ambienti relativamente liberi. Il linguaggio è fondato su regole, con la selezione della regola basata sull'informazione percepita dall'ambiente combinata con l'informazione che descrive la direzione attuale del robot e la conoscenza che riguarda l'ambiente locale.
Le regole di Altaira sono divise in due parti: nella parte superiore ci sono le precondizioni e nella parte inferiore ci sono le azioni o le postcondizioni.
Le regole hanno quattro condizioni di input e quattro azioni di output. Le quattro condizioni di input sono: lo stato dei sensori, la direzione corrente, lo stato attuale della mattonella e lo stato del robot. Per ogni input, o è richiesta una corrispondenza esatta oppure la regola può specificare quell'input come *dontcare*. Le quattro azioni di uscita sono: i comandi ai motori, il controllo della navigazione, il nuovo stato della mattonella e il nuovo stato del robot.
Tutte le regole sono conformi ad un modello standard che ne facilita il processo di sviluppo; inoltre le regole hanno una priorità dovuta alla possibilità che molte regole siano eseguite in ogni ciclo di esecuzione.



Rule editor

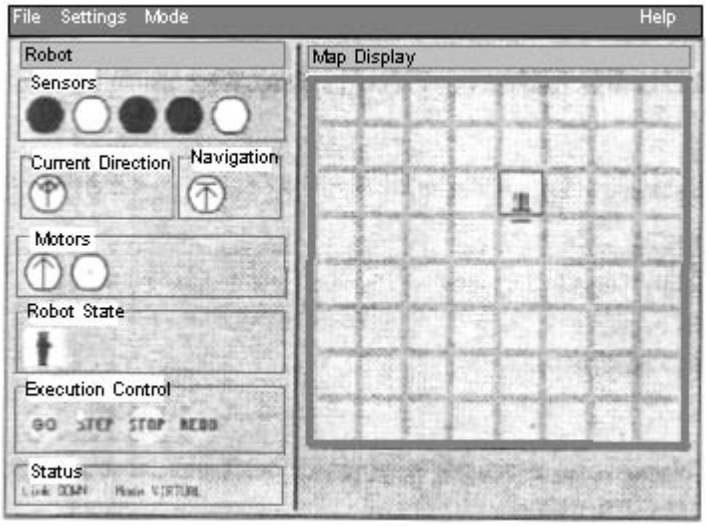
Il ciclo di esecuzione è basato sull'interazione di due strutture dati: un array bidimensionale di stati della mattonella chiamato *mappa* e un insieme ordinato di regole chiamato *ruleset*.
In ogni ciclo di esecuzione, sono eseguiti i seguenti passi:

1. Sono testati i sensori del robot.
2. Sulla base dei sensori, lo stato attuale della mattonella e lo stato del robot, una o più regole sono abilitate per essere eseguire.
3. L'insieme delle regole permesse sono combinate per produrre i nuovi valori per gli attuatori del robot, lo stato della mattonella e lo stato del robot.

Quando più di una regola è permessa in un ciclo di esecuzione, le regole sono combinate (il termine usato in Altaira è "amalgamate").
Concettualmente, a tutte le regole permesse è assegnata una priorità basata sul fatto che abbiano input specificati o dontcare, nel seguente ordine:

1. Le regole con un input specificato per quanto riguarda lo stato del robot, hanno una priorità più alta di quelle con un dontcare. Queste regole sono chiamate *regole strategiche,* poiché sono usate generalmente nell'implementazione di subgoal strategici.
2. Le regole con un input specificato per quanto riguarda lo stato della mattonella, hanno una priorità più alta di quelle con un dontcare. Le regole con lo stato della mattonella (ma non lo stato del robot) specificato sono chiamate *regole tattiche,* poiché sono usate nell'implementazione di tattiche locali per una particolare mattonella.
   Regole che non specificano o lo stato del robot o quello della mattonella sono chiamate *regole riflessive*, poiché sono semplici risposte a stimoli senza riguardo per lo stato.
3. Le regole con una direzione specifica hanno una priorità più alta di quelle con un dontcare.
4. Le regole con più sensori di input specificati hanno una priorità più alta di quelle con meno. La priorità è arbitraria fra le regole con lo stesso numero di sensori di input specificati.
5. La regola amalgamata che risulta dal precedente ciclo di esecuzione è la regola con priorità più bassa (regola di default).

Le regole permesse sono considerate nell'ordine da quella con più bassa priorità a quella con priorità più alta. Mentre una regola è considerata, i suoi output sostituiscono i valori nella regola amalgamata.
Questa unione di regole implementa un'architettura di tipo subsumption.
Nella maggior parte dei cicli di esecuzione, sono applicate soltanto le regole riflessive. Occasionalmente, è applicata una regola che dipende dallo stato attuale della mattonella; l'output di questa regola tattica sostituisce l'output che sarebbe fornito da una regola riflessiva.
Per concludere, in circostanze molto rare, una regola strategica esclude il risultato delle tattiche o delle riflessive.
Ci sono due meccanismi per terminare l'esecuzione. In primo luogo, esiste un meccanismo per determinare se ci sia ancora del lavoro da svolgere nell'ambiente. Quando gli stati delle mattonelle sono creati, l'utente può dichiararli o terminali o non terminali. In ogni ciclo di esecuzione, il sistema esamina lo stato di ogni mattonella nella mappa e determina se ce ne sono di non terminali. Se non ci sono stati non terminali, l'esecuzione è terminata.
In secondo luogo, esiste un meccanismo che determina se un obiettivo è stato raggiunto. Le condizioni di terminazione esistono sia per gli stati del robot che delle mattonelle. Se le mattonelle o il robot sono in stato di terminazione, l'esecuzione si arresta immediatamente.[7]
L' "execution engine" implementa il ciclo di esecuzione descritto sopra e si occupa di mostrare le informazioni che riguardano l'attuale stato del sistema (includendo la mappa delle mattonelle, il robot e i link di comunicazione).



Execution Engine

Un editor permette all'utente di definire regole in modo completamente grafico (rule editor). Sono definite delle icone per gli input relativi ai sensori e alla direzione, e per gli output inerenti ai motori e alla navigazione. L'utente è in grado di navigare nel ruleset selezionando l'input della regola (ogni volta che un nuovo input di una regola è selezionato, l'appropriata regola è localizzata nel ruleset e il suo output è mostrato) e può modificare la regola corrente selezionando un nuovo valore per il suo output.
I menu legano gli input e gli output delle regole permettendo all'utente di fare delle scelte a seconda delle sue esigenze. Il menu relativo allo stato, permette di invocare l'editor di stato che consente di definite nuovi stati.
L'utente è in grado di definire icone per lo stato della mattonella usando un semplice editor di icone. Uno stato include l'icona definita con questo editor, un indice assegnato dal sistema e un flag che permette all'utente di settare lo stato come terminale o non terminale.[9]
Una descrizione più dettagliata del linguaggio applicato alla risoluzione della sfida può essere trovata in [2], [7], [8], [9].

RoadSurf

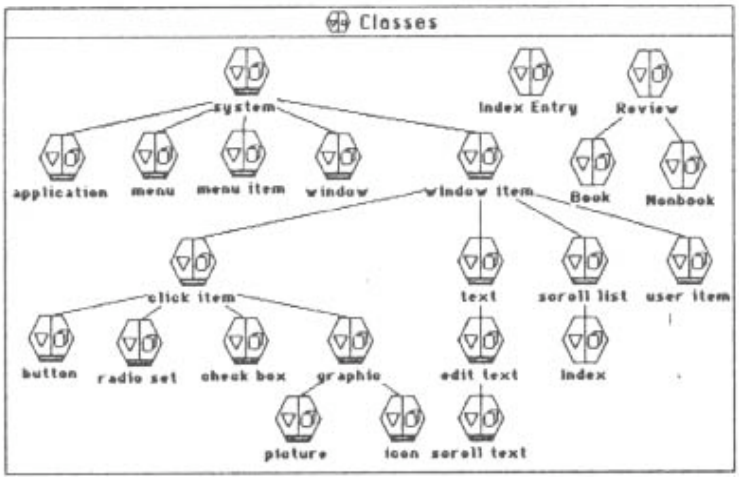
RoadSurf è un linguaggio di programmazione rule-based, nel quale l'utente definisce comportamenti da dichiarare in forms. Questi comportamenti mappano gli eventi rilevati dai sensori in azioni.[2]

Linguaggi di tipo Dataflow

Prograph

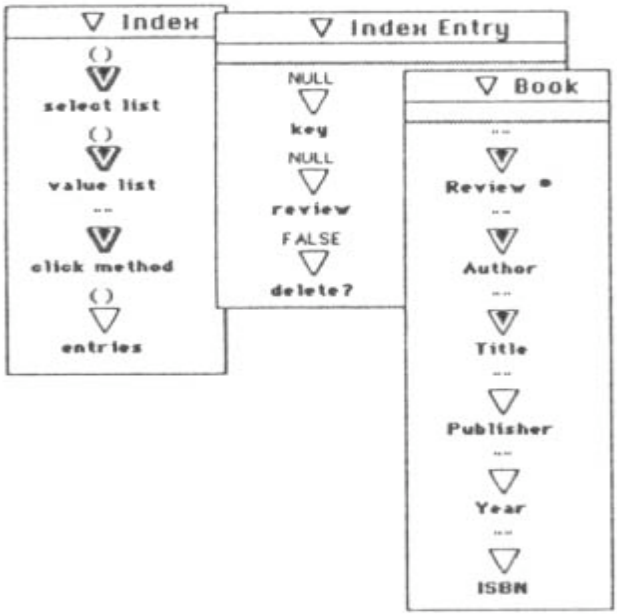
Prograph è un linguaggio di programmazione object-oriented visuale. Combina il modello object-oriented con il modello dataflow, risultando un approccio che qualche volta viene definito *object-flow*.
Si descrive ora come Prograph implementa i concetti principali della programmazione orientata agli oggetti: *classi*, *attributi* e *metodi*.
Una *classe*, nella programmazione orientata agli oggetti, fornisce un meccanismo per la definizione di un tipo di dato, con attributi e metodi che possono essere applicati agli oggetti di questo tipo.

Le classi in Prograph sono organizzate in gerarchie: la struttura delle classi di un programma Prograph è mostrata nella finestra *Classes*. Questa finestra contiene una rappresentazione visuale dell'albero attuale delle classi. Ogni classe, è rappresentata da un'icona e ogni relazione tra classi padre-figlio è indicata con una linea che va dall'icona del padre a quella del figlio.



Esempio di classi

Ci sono due tipi di classi in Prograph: classi di sistema e classi utente. Prograph ha una predefinita gerarchia di classi di sistema che fornisce strutture per l'interfaccia con l'utente come finestre, menu, finestre di dialogo, pulsanti e liste per un'applicazione Prograph. Un'istanza di una classe di sistema avrà attributi di sistema e un appropriato comportamento quando l'applicazione Prograph è in esecuzione. Le classi di sistema sono distinte da quelle utente da una doppia barra sulla parte bassa dell'icona della classe.
Torniamo alla finestra *Classes*. L'icona di una classe contiene due più piccole icone che rappresentano *attributi* e *metodi*, i componenti di una classe. Ogni classe eredita tutti gli attributi e i metodi dalle classi progenitori e può avere metodi e attributi addizionali. Gli attributi di una classe sono rappresentati in una finestra detta appunto *finestra degli attributi*.
Una classe può avere attributi di classe, che sono invarianti per tutte le istanze della classe, e attributi di istanza, che possono avere distinti valori per istanze singole.



Esempio di attributi

All'interno di una finestra relativa agli attributi, una linea orizzontale separa gli attributi di classe da quelli di istanza. Un attributo ereditato da un progenitore è indicato da punta di freccia nell'icona dell'attributo. Gli attributi di sistema sono indicati con un'ulteriore linea sui lati dell'icona dell'attributo. Ogni attributo ha un valore di default che è usato per inizializzare una nuova istanza della classe ed è mostrato sopra l'icona dell'attributo.
Un *metodo* in Prograph è di classe o universale, appartenente a nessuna classe. I metodi delle classi sono rappresentati da icone in una *finestra dei metodi* per le classi e i metodi universali definiti dall'utente sono rappresentati da icone nella finestra *Universal*.
Un metodo è costituito da una sequenza di *cases*, dove ogni case è una struttura dataflow composta da input e output di dati, un insieme di operazioni e collegamenti tra questi.
La definizione di un case è illustrata graficamente in una finestra per il case. L'input nel case è indicato da *roots* su una barra di input in cima alla finestra del case, e l'output da *terminals* sulla parte bassa della barra di output. L'arietà di input e di output di un case è il numero dei suoi roots e terminals rispettivamente e tutti i case di un metodo hanno la stessa arietà di input e di output. Un'operazione in un case è rappresentata da un icona che contiene il nome dell'operazione.
L'input e l'output di dati per un'operazione sono definiti da roots e terminals in cima e in fondo all'icona dell'operazione. Le operazioni nel case sono connesse da *data link* che portano i dati dai terminals delle operazioni ai roots delle altre operazioni. Un'operazione è analoga a una chiamata a procedura e terminals e roots per un'operazione si comportano come i parametri di una procedura. Comunque, diversamente dalla maggior parte dei linguaggi di programmazione testuali, i valori dei dati non hanno un tipo e un terminal o un root di un'operazione rappresenta l'azione di copiare il valore di un dato tra l'operazione chiamata e il metodo associato. Un root può essere attaccato a diversi data links, ma un terminal può essere connesso al massimo a uno. L'arietà di un'operazione deve essere la stessa definita nel metodo associato.
Procediamo ora con una breve rassegna sulle diverse operazioni disponibili in Prograph. Queste operazioni sono rappresentate da icone la cui forme rispecchia il tipo dell'operazione stessa.

Input:	copia valori dai terminals dell'operazione chiamata
Output:	copia i valori nei roots dell'operazione chiamata
Simple:	chiama un metodo definito dall'utente o una primitiva di Prograph
Constant:	è etichettata dalla costante che è prodotta come output. Il tipo di costante deve essere uno di quelli incorporati in Prograph cioè boolean, integer, liste e stringhe.
Persistent:	permette di accedere a strutture per l'immagazzinamento di dati
Instance:	genera una nuova istanza di una classe. Normalmente i valori degli attributi della nuova istanza saranno quelli di default
GetAttribute:	riceve in input un'istanza e restituisce l'istanza e il valore dell'attributo dell'istanza di input con cui è etichettata
SetAttribute:	riceve in input un'istanza e un valore da attribuire all'attributo con cui è etichettata. Restituisce una copia dell'istanza con l'attributo con cui è etichettata settato al valore di input.
Local:	chiama un metodo localmente definito

L'esecuzione di un metodo Prograph inizia con una chiamata al metodo e con il passaggio di dati di input agli input roots. Il primo case della sequenza di case del metodo inizia l'esecuzione. L'esecuzione di un case segue il data driven, il paradigma dataflow determinato dalla rappresentazione grafica del case. Infatti, un'operazione con degli input non è chiamata nell'esecuzione di un case finché riceve tutti i suoi dati di ingresso e operazioni per cui non ci sono interdipendenze sono considerate logicamente eseguite in parallelo. L'output da un case è disponibile solo quando l'esecuzione del case termina. Normalmente, un case non termina finché tutte le operazioni all'interno del case sono state eseguite.
Un commento può essere annesso ad ogni icona o elemento di programmazione e serve solo a fornire informazioni addizionali.

Iterazione e parallelismo sono comuni paradigmi di programmazione e Prograph fornisce un ricco insieme di meccanismi di iterazione e parallelismo attraverso *multiplexes*.
Una lista, uno dei tipi di dati fondamentali in Prograph, possono essere processati usando vari multiplexes diversi. Il multiplex accetta in input una lista, applica l'operazione a ogni elemento della lista e assembla i risultati ancora in una lista. Gli elementi della lista possono essere quindi processati in parallelo.
Un altro multiplex associato con il processamento di liste è il *partition multiplex*, che applica un'operazione booleana a ogni elemento di una lista di input. L'output consiste di due liste: una degli elementi per i quali l'operazione booleana ha prodotto TRUE e una per i quali l'operazione booleana ha prodotto FALSE.
Infine diamo un breve sguardo all'ambiente di sviluppo di Prograph.
L'ambiente Prograph ha tre componenti fondamentali, l'*editor*, l'*interprete* e l'*application builder.*
L'editor è usato per il progetto e la costruzione di programmi, l'interprete esegue un programma e fornisce utili funzioni per il debugging e l'application builder semplifica il compito di costruire un'interfaccia grafica per il programma.[10]

Una descrizione più dettagliata del linguaggio applicato alla risoluzione della sfida può essere trovata in [2], [4], [5], [6]. In questi esempi applicativi, sono stati usati diversi approcci.
Il primo affronta il problema seguendo un taglio proprio della programmazione visuale procedurale. Sono state sviluppate un insieme di funzioni di basso livello, per la comunicazione con l'hardware del robot, e alcune di livello più alto per aumentare il grado di astrazione relativamente a questioni inerenti all'hardware.
Il secondo approccio segue un taglio proprio della programmazione visuale object-oriented. In questo ambito, uno degli obiettivi era generalizzare il programma per altre configurazioni del robot oltre che agevolare la programmazione con tecniche visuali.

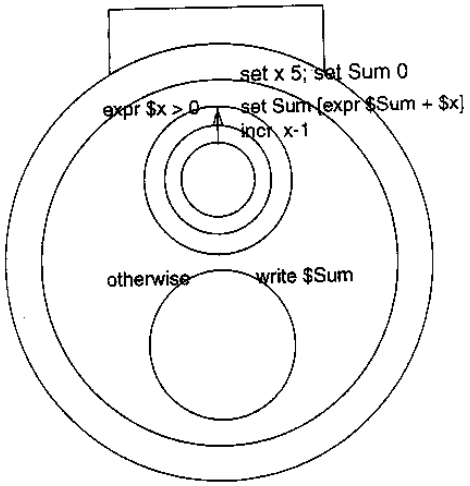
Linguaggi di tipo Controlflow

VIPR

Introduzione.
Il VIPR [12],[13],[10] è un linguaggio visuale imperativo, composto da un piccolo numero di componenti visuali. Queste componenti sono supportate dall'utilizzo del linguaggio Tcl per gli script a loro connessi. Il VIPR ha punti di forza e punti deboli dei linguaggi testuali. I punti di forza sono l'astrazione , la familiarità e la flessibilità; i punti deboli dei linguaggi visuali imperativi sono da ricercarsi nella mancanza di una rappresentazione grafica esplicita e dalla impossibilità di avere contemporaneamente una visione globale ed una locale del programma.
Con VIPR si vuole portare la forza dei linguaggi imperativi tradizionali nel contesto dei linguaggi visuali. Lo scopo è, quindi, quello di rendere il programmare più facile e di minimizzare il numero di dipendenze nascoste.
Il VIPR supporta, inoltre, un numero di funzioni per facilitare la fase di scrittura e di debug di un programma. Consente l'esecuzione animata e risolve il problema della scalabilità attraverso l'utilizzo delle tecniche di zooming e fish-eyeing[14].

VIPR è un linguaggio grafico, sviluppato per evitare i problemi di scalabilità e di dettaglio, incontrati negli altri linguaggi visuali.
E' un linguaggio imperativo che usa il linguaggio Tcl per gli script e costrutti visuali per il controllo, comprendendo le procedure. VIPR unisce la facilità di programmazione, ed anche la flessibilità di un linguaggio che utilizza gli script, con la chiarezza della semantica dei linguaggi visuali e la possibilità di ampliamento dei linguaggi procedurali.

VIPR ha poche primitive grafiche che sono usate per comporre un programma. Queste primitive sono cerchi, frecce e cerchi con parametri. Un esempio dei concetti espressi lo vediamo in Figura v1. Il programma somma i numeri da 5 a 1 e fornisce il risultato.




```
set x 5; set Sum 0
while [expr $x > 0] {
   set Sum [expr $Sum + $x]
   incr x - 1
}
write $Sum
```

Figura v1: un esempio di loop in Vipr ed il suo equivalente in Tcl.

Il primo passo (cerchio più esterno) setta il valore della variabile "x" a 5 e setta "Sum" a 0. Il controllo passa ora ai due cerchi nidificati dentro il cerchio più esterno. Ogni cerchio ha una espressione di guardia sulla sinistra e una azione sulla destra. Una espressione di guardia vuota o non esistente equivale al soddisfacimento della condizione (cioè è vero). VIPR valuta l'espressione di guardia, in questo esempio "expr$x>0" e trova che è vera. Allora esegue l'azione descritta sulla destra ("set Sum[expr$Sum+$x]") che somma il valore di x a Sum attuale. Successivamente, il valore di x viene decrementato ("incrx-1"). Il controllo torna, ora, al ciclo "while" (indicato dalla freccia sul cerchio con l'espressione $x>0 guard). Questo ciclo viene eseguito fino a quando la guardia non è più vera. Quando questo capita, il cerchio marcato "otherwise" viene eseguito e fornisce il risultato di "Sum".
Le primitive del VIPR possono essere combinate per scrivere programmi di qualsiasi complessità. Le frecce vengono utilizzate per trasferire il controllo tra le varie sezioni del programma.
La semantica del VIPR è basata sui principi di sequenza, di esecuzione sotto guardia e di sostituzione. Il concetto di sequenza si rispecchia nella sintassi del controllo; l'approccio a cerchi nidificati determina la sequenza di esecuzione. Come il programma viene eseguito, ogni cerchio viene successivamente incorporato nel cerchio più esterno. Questo approccio può essere riassunto come "segui i cerchi più profondi nel gruppo".
L'esecuzione sotto guardia è il metodo usato dal VIPR per mostrare l'esecuzione sotto condizione. Ogni cerchio può avere una guardia (posizionata nella porzione in alto a sinistra del cerchio) e una espressione per l'esecuzione (posizionata nella porzione in alto a destra del cerchio). Ogni guardia ha un valore booleano.
La sostituzione è l'altro elemento per il controllo semantico del VIPR. Le frecce vengono utilizzate al fine di collegare le sezioni del programma. Inoltre, le frecce determinano la sezione del programma che sarà sostituita durante l'esecuzione. La semantica del VIPR può essere riassunta nei seguenti punti:

- segue i cerchi in profondità nelle nidificazioni (come nel C si seguono le righe);
- se c'è una guardia, si esegue l'azione solo se il valore booleano è vero;
- se c'è una freccia, il controllo la segue.

Poiché anche i piccoli programmi possono richiedere nidificazioni complesse, questo può costituire un problema per la leggibilità di un programma in VIPR, se non c'è il supporto per lo zooming. Lo zooming permette, infatti, al programmatore di specificare un frammento di codice e di allargarlo per poterlo leggere meglio e modificarlo. Il programmatore può rimpicciolire ed ingrandire il programma a piacere.

Ambiente di sviluppo del VIPR:

L'interfaccia standard dell'ambiente di programmazione in VIPR è presentato in fig.v2.

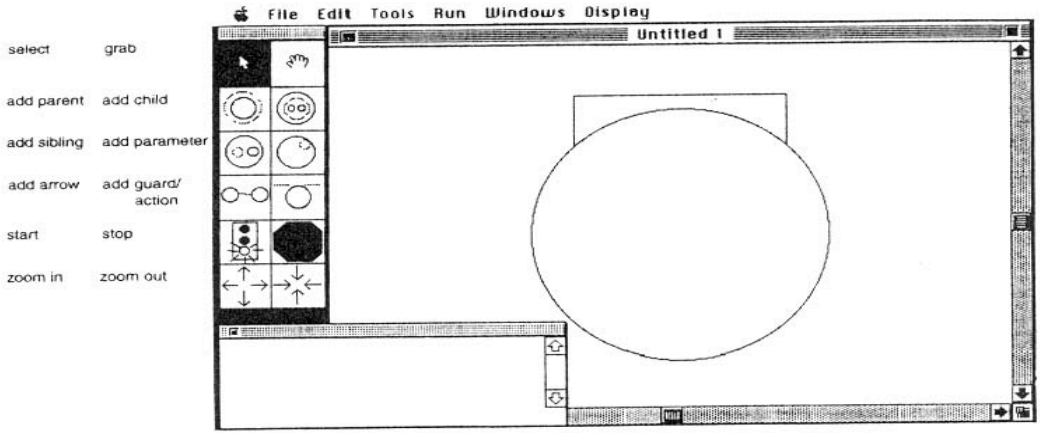


Figura v2: esempio di ambiente di sviluppo in Vipr.

La voce "select" premette di selezionare oggetti per cancellare o per duplicare. La voce "grab" premette di spostare i cerchi; le altre voci consentono varie azioni, tra cui creare cerchi all'esterno (parent ring) oppure all'interno (sibling ring) di un cerchio, introdurre guardie, frecce e così via.
L'utilizzo delle nidificazioni consente di ridurre il numero di linee che si intersecano (problema importante dei linguaggi control-flow); tuttavia, se le nidificazioni vanno troppo in profondità, risulta impossibile la lettura del programma. Per risolvere questo problema, il VIPR supporta le tecniche di zooming e di fish-eyeing. Lo zooming, come visto in precedenza, permette di focalizzare una parte del codice, ma si perde il contesto di quella parte all'interno del programma. Per le situazioni in cui si preferisce avere una visione precisa di una parte del codice, ma anche una visione del contesto, il VIPR utilizza una visione fish-eye, basata sulla proiezione sferica. Il VIPR permetta anche l'animazione run-time del programma, mostrando le frecce che vengono seguite dal controllo e le sostituzioni eseguite; può anche essere eseguito un cerchio alla volta e l'esecuzione può essere interrotta in qualsiasi punto e ripresa dopo aver fatto o no modifiche al programma.

Progetto al VPC:

L'architettura ad alto livello della soluzione del VIPR al VPC è mostrata in figura v3.

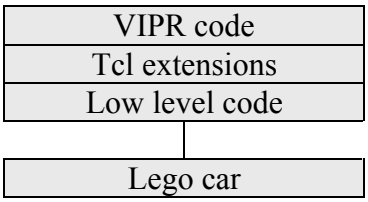


Fig.v3: architettura ad alto livello della soluzione del VIPR al VPC.

La struttura è la tipica tecnica top-down dei linguaggi imperativi e permette il riutilizzo del codice. Le varie procedure scritte sia in VIPR che in Tcl sono estensioni del C e riutilizzabili. Esiste quindi la possibilità di sviluppare librerie in VIPR.
Il vantaggio del VIPR rispetto agli altri linguaggi imperativi è il modo con cui vengono indicate le procedure, che, come già visto, consiste in una semplice freccia. Le frecce, infatti, denotano i collegamenti tra le varie procedure. Questa rappresentazione consente una "mappa" del codice ed è più facile seguire la sua esecuzione.

Implementazione:

L'attuale soluzione ruota attorno alla decisione di delimitare il confine tra il codice per il supporto a basso livello e il codice VIPR. Le funzioni a basso livello si occupano dei comandi per il veicolo e viene fornita una semplice interfaccia per chiamare queste funzioni. Queste funzioni vengono tradotte in Lisp e in C, così che il VIPR le possa chiamare come Tcl. Il Tcl ha un semplice struttura per scrivere velocemente estensioni del linguaggio. Questi ampliamenti sono scritti in C e quindi chiamati come funzioni Tcl. Tutte le interazioni sono fatte tra il VIPR e il Tcl o tra il Tcl e il C. Il VIPR chiama una estensione Tcl che conta sul sottostante codice C per eseguire il comando richiesto.
La possibilità di utilizzare e creare rapidamente l'estensioni in Tcl fornisce al VIPR di stabilire un confine flessibile tra il codice di supporto e il VIPR. Questo confine viene aggiustato a seconda della applicazione e viene scelto dal programmatore e non imposto dal linguaggio.
La decisione di dove posizionare il confine tra il VIPR e le estensioni Tcl ha un forte effetto sulla soluzione complessiva del problema del VPC. Determina, infatti, quante informazioni il programmatore necessita di sapere prima di scrivere il codice. La decisione di limitare la quantità di codice di supporto alla sola manipolazione diretta della macchina, è dovuta alla volontà di misurare la flessibilità del VIPR.
Per capire la soluzione del VIPR cominciamo dalla figura v4.

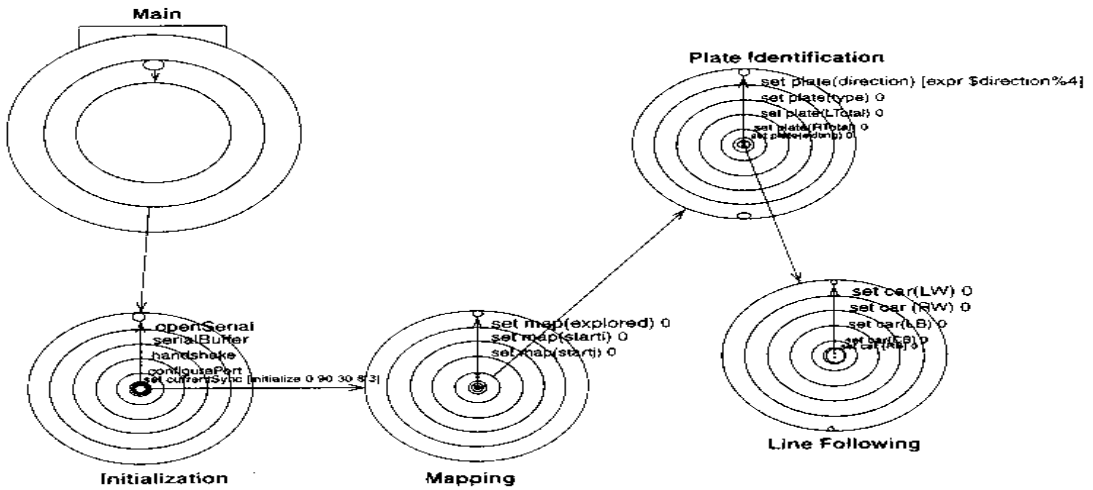


Figura v4: procedure della soluzione del Vipr al VPC.

La procedura di inizializzazione setta semplicemente un numero di parametri tra cui la soglia per differenziare il nero ed il bianco e il numero dei comandi attuali. Questi parametri vengono passati, attraverso una estensione Tcl, al codice di basso livello che li manipola per fornirli all'interprete run-time e per mandarli alla porta seriale del Handy Board.
In modo simile, i comandi attuali per muovere il veicolo tramite i motori e i comandi per avere i valori dei sensori sono chiamati solo nella routine Line Following. Ciò porta ad una architettura in cui solo Inizialization e Line Following chiamano il codice di basso livello per manipolare la macchina.

Solo con una eccezione (le variabili globali), l'informazione che è presente in una procedura non è accessibile dalle altre. Questo non capita tra Plate Identification e Line Following, la cui ragione sarà discussa dopo.
La soluzione del VIPR è costituita da 4 procedure.

- Inizialization manipola il setup della macchina e la porta seriale.
- Mapping crea e mantiene la mappa esplorata; si occupa inoltre del corretto posizionamento delle piastre stradali nella mappa.
- Plate Identification tiene traccia delle informazioni del tipo indicatori di svolta, che sono usati per determinare che tipo di piastra è stata attraversata e il corrente orientamento della macchina.
- Line Following si occupa della lettura dei sensori e dell'azionamento dei motori. Quando Line Following incontra situazioni che richiedono conoscenza circa la piastra, l'informazione viene ripassata a Plate Identification.

La soluzione VIPR ha, quindi, procedure che compiono decisioni run-time. Queste sono Plate Identification e Line Following. La gestione della mappa è affidata alla procedura mappa. Prende informazioni dalla procedura Plate e tiene semplicemente traccia del tipo di piastra, la sua posizione nella griglia e le piastre non esplorate adiacenti a quelle esplorate. La procedura può essere ben più intelligente, dal momento che in molti casi è possibile determinare il tipo di piastra non esplorata, conoscendo la tipologia di quelle adiacenti. Tale procedura usa una funzione display basata su Tk per mostrare la mappa fino in quel momento riconosciuta.
Precedentemente al VPC, il VIPR aveva scarse funzioni di display. VIPR ha un piccolo display che mostra i valori delle variabili, ma non grafici. Può anche mostrare input e output, ma sempre con caratteri ASCII. Per risolvere questo problema è stato aggiunto un display Tk. Questo compie un processo a parte e riceve i comandi attraverso un socket con il VIPR. Questa semplice estensione Tk manda una bitmap della piastra in una cerca posizione e con un certo orientamento. Questo permette al programma di mostrare l'informazione necessaria senza entrare in conflitto con i loop del VIPR.

Procedure Line Following e Plate Identification:

Line Following è una semplice procedura che ha il compito di far seguire una linea base tramite le informazioni fornite dai sensori. Esegue, attraverso i motori, semplici correzioni nelle dovute direzioni. Conta sul fatto che il sensore centrale debba leggere nero, dal momento che il nero denota il centro della strada. Poiché i motori fanno spostare di poco, è possibile correggere la direzione basandosi sull'assenza della lettura del nero sul sensore centrale e sulla presenza del nero sul sensore di destra o di sinistra. Questo approccio è semplice, ma può portare ad una rappresentazione confusa della procedura. Nella situazione in cui tutti i sensori leggono nero, Line Following assume che sia stata trovata la fine di una piastra. In molte situazioni, la procedura Line Following restituisce il controllo alla procedura Plate Identification, che esamina i dati passati.
Questa struttura tiene traccia dei segnali di svolta individuati tramite i sensori ai lati e dove sia possibile o no riconoscere la fine della piastra.
Per esempio, un veicolo che attraversa una intersezione + dovrebbe vedere due insiemi di segnali di svolta prima di lasciare la piastra. Il segnale di svolta avvistato può essere differenziato dal centro del marker piastra + dalla combinazione di "distanza del viaggio" e dal fatto che tutti i sensori leggeranno nero appena il veicolo attraversa il centro della piastra. Tale ragionamento può essere applicato anche mentre la macchina svolta.
Plate Identification determina se la piastra è già stata visitata; se no, chiama le procedure Line Following e così via.
Analizziamo in modo più dettagliato la procedura Line Following. La figura v5 mostra il punto di ingresso alla procedura.

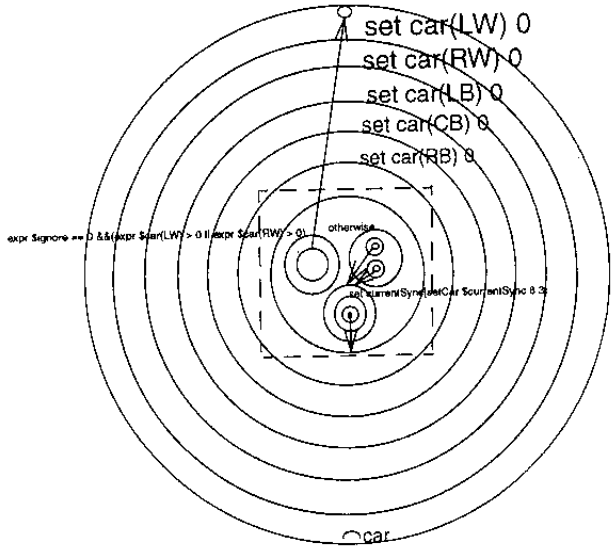


Figura v5: il punto di ingresso alla procedura Line Following.

Ci sono due cerchi parametrici, uno per tornare a Plate Identification e l'altro per passare la struttura $car dentro e fuori da Line Following.
Innanzitutto il codice inizializza i valori in $car a 0. I comandi mostrati (set car(LW) 0, etc...) espongono il modo con cui in Tcl si costruiscono le strutture dati. Set car(LW) 0 setta LW(sensore di sinistra) di $car a 0.
La figura v6 è un ingrandimento del quadrato tratteggiato in figura v5 e si può analizzare il ciclo while.

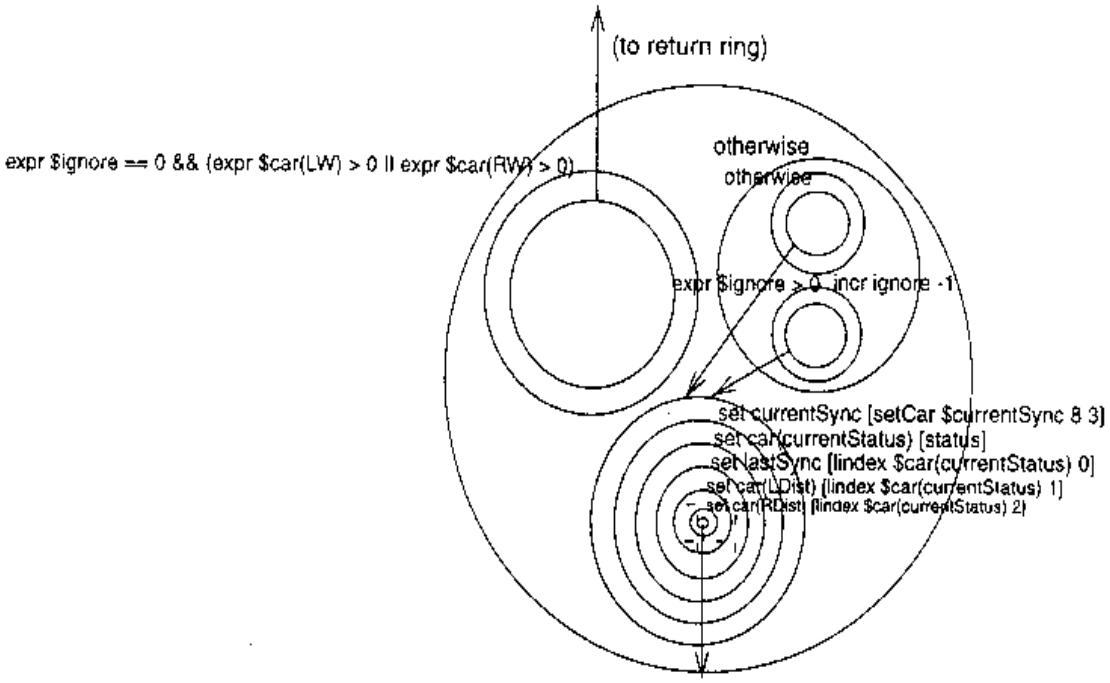


Figura v6: il sensore corrente viene letto nella struttura $car.

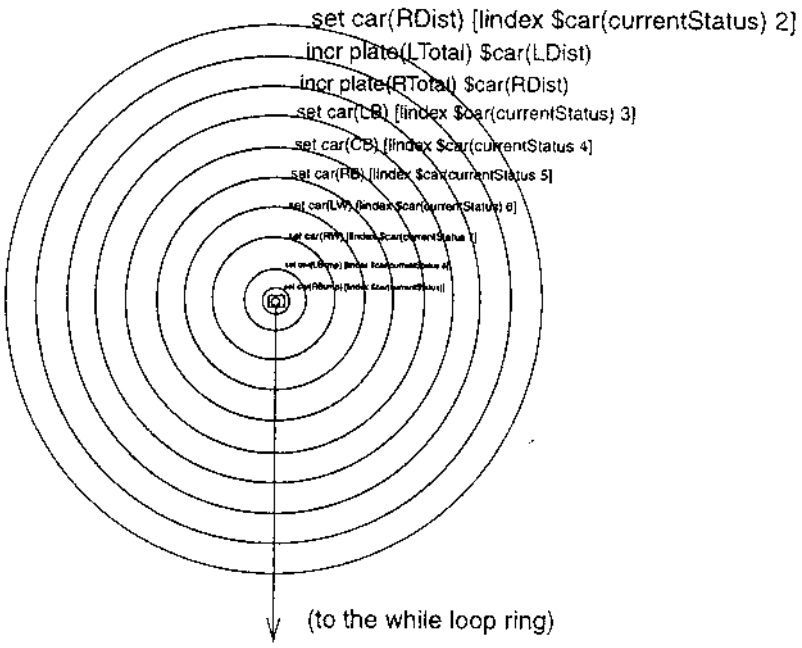


Figura v7: codice per riempire la struttura dati con l'array dei valori che rappresentano lo stato della macchina.

Il cerchio più esterno tratta il caso in cui Line Following non può determinare l'azione da compiere e deve restituire il controllo a Plate Identification. L'espressione $ignore==0 permette di simulare uno stato "don't care". L'insieme di cerchi sulla destra in alto sarà eseguito se $ignore==0 o se nessuno dei due sensori è on. Ignore viene decrementato di 1 se è più grande di 0 e il controllo passa ai cerchi più interni. Questo è un ciclo nidificato if-then-else. I cerchi più interni iniziano il lavoro per manovrare la macchina. Il numero di sincronizzazione è ottenuto interrogando la macchina con l'estensione Tcl setCar. In certi casi il numero di sincronismo è necessario per assicurarsi che un dato comando venga eseguito.
L'estensione Tcl sono fatte a basso livello e questo si riflette nella conoscenza che l'utente ha della macchina. In questo caso, il comando prende tre argomenti, il numero di sincronizzazione, la velocità e la massima distanza.
La figura v7 presenta il codice per riempire la struttura dati con l'array dei valori che rappresentano lo stato della macchina. In figura v8 si vede il codice per guidare la macchina e presenta una struttura if-then-elseif-else:

- se il sensore di destra è on ($car(RB)>0), la macchina è guidata verso sinistra;
- se il sensore di sinistra è on ($car(LB)>0), la macchina è guidata verso destra;
- se nessuno dei due è on, entrambi i motori vengono attivati.

Riassumendo , Line Following effettua solo le funzioni per muovere la macchina; le decisioni che richiedono la conoscenza di alcune informazioni, vengono prese in Plate Identification.

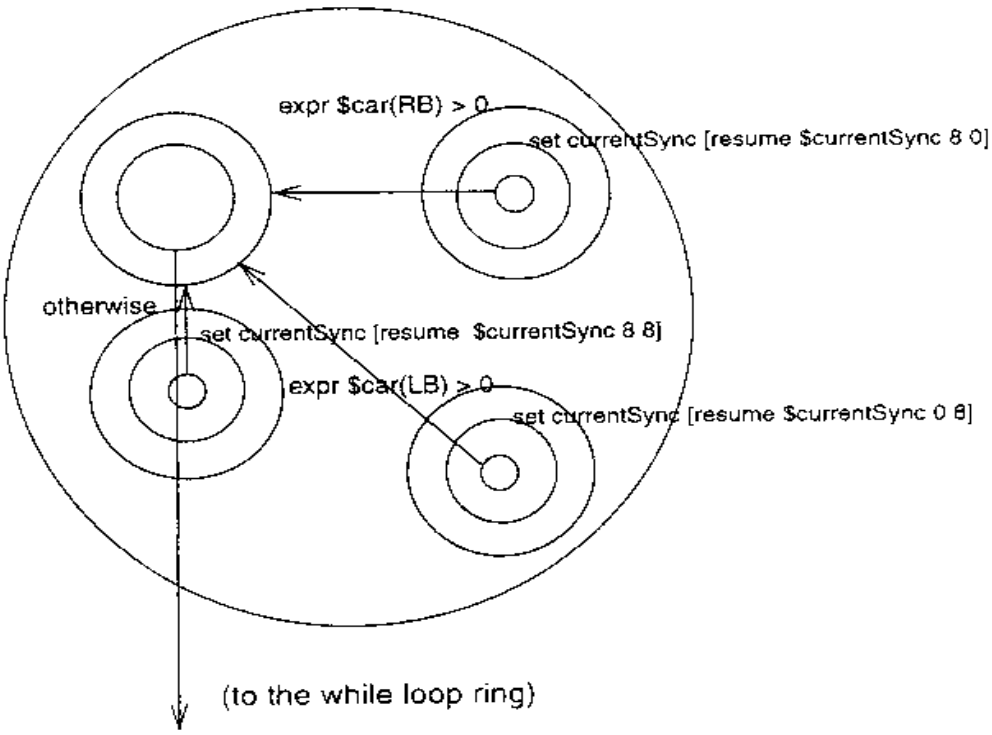


Figura v8: il codice in figura chiama il comando resume, che causa l'avviamento dei motori della macchina.

La figura v9 mostra una sotto sezione del codice di Plate Identification. Il codice in questione esamina il valore attuale di $plate (leftIndicator) per vedere quanti indicatori di svolta a sinistra sono stati incontrati. Ognuno dei cicli if-then-else aggiornano la struttura $plate e settano la variabile ignore, con il numero di impulsi da ignorare. Questa variabile premette alla procedura Line Following di ignorare il valore dei sensori laterali per un certo numero di impulsi che consentono di trascurare la piastra segnata.

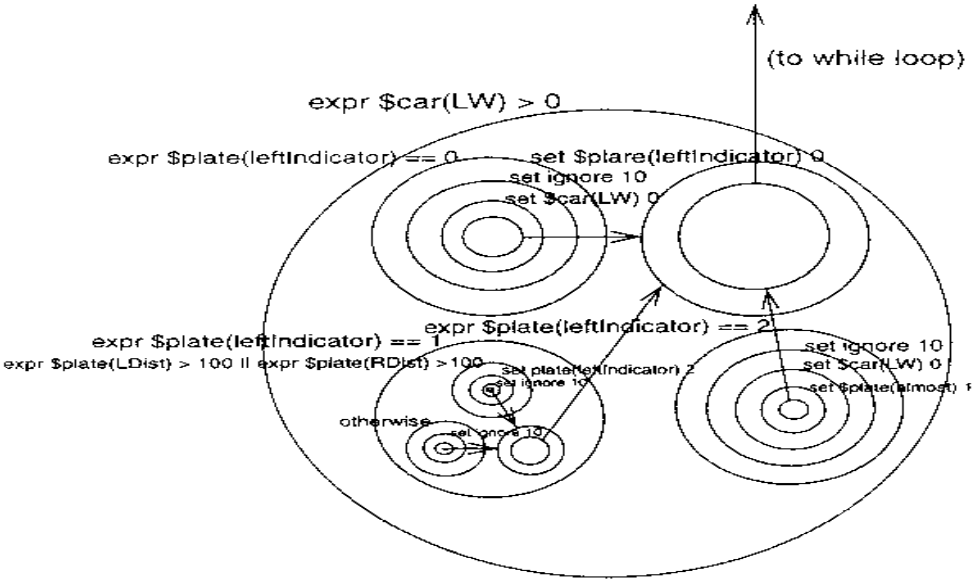


Figura v9: questa piccola sezione della procedura Plate Identification analizza il caso in cui il sensore sinistro è acceso.

Richieste non funzionali:

Le richieste non funzionali del VPC hanno svolto un importante ruolo nella implementazione della soluzione. Per esempio, un problema non considerato come parametro di valutazione è stato l'importanza di ridurre il tempo di latenza tra quando un comando viene dato e quando esso viene eseguito. Il peso su questo parametro è dato dall'animazione dell'esecuzione del programma. Con l'animazione si perdono 30 secondi prima di azionare i motori e senza l'animazione poco più di un secondo. VIPR supporta la richiesta di flessibilità e il riutilizzo del codice senza modificare il codice esistente. Tuttavia anche altre caratteristiche, quali la scalabilità ed altre in cui il VIPR si esprime al meglio non sono state oggetto di valutazione al VPC. Viceversa l'aspetto pedagogico, che nel VIPR non è un obiettivo , è stato un criterio fondamentale.

Create

Create è un linguaggio controlflow che utilizza diagrammi di flusso, molti usano cerchi annidati come VIPR, con lo scopo di rappresentare il flusso di controllo.[2]

Cwave

Cwave è un linguaggio di programmazione controlflow di tipo box-and-arrow. Il linguaggio base è stato esteso per trattare specificamente con l'interfaccia del mini-calcolatore montato sul robot. Gli autori lavorano su un ambiente generale "nel quale un dominio di progetto sviluppa componenti di un dominio specifico". Uno di questi nuovi componenti si mostra come un'icona del veicolo, i suoi sensori e motori. Con questo componente essenziale gli utenti possono definire modelli descrivendo i valori dei sensori come condizioni e i valori dei motori come azioni. Quando le condizioni sono soddisfatte, i motori sono settati ai loro nuovi valori e il flusso di controllo continua.
La natura del flusso di controllo conduce a una concettualizzazione della soluzione che è abbastanza differente dagli approcci rule-based. In maniera molto diversa dalle regole, i componenti usati sono cambiati insieme esplicitamente specificando l'ordine. Un inconveniente di questa soluzione è che i componenti sono implementati in linguaggi di programmazione convenzionali, come C++, e quindi uniti insieme in un diagramma controlflow, con un'evidente confusione.[2]

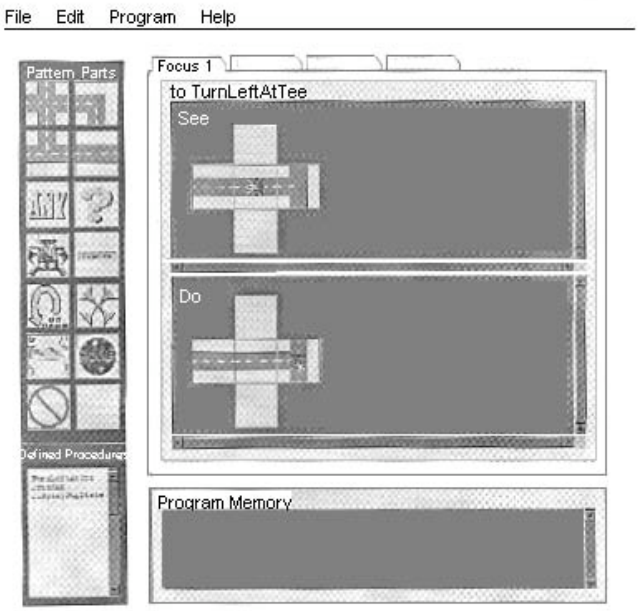
SeeDo

SeeDo è un linguaggio di programmazione visuale progettato per fare fronte alle richieste delle 1997 Visual Programming Challenge. E' un linguaggio ibrido, avendo le caratteristiche per supportare sia programmazione visuale che testuale, così come programmazione rule-based e procedurale.
Le caratteristiche del linguaggio sono orientate soprattutto per l'uso da parte di programmatori non professionisti. Questo orientamento primario non preclude l'uso di SeeDo a programmatori con esperienza. I progettisti di SeeDo ritengono che un linguaggio con un semplice nucleo dei concetti, più alcune caratteristiche avanzate fondamentali, possa essere usato efficacemente da programmatori a vari livelli di competenza.

L'ambiente di sviluppo di SeeDo è costituito dei seguenti componenti principali:

- Una o più finestre per editare programmi. Ciascuna di queste finestre è chiamato *focus* e incorpora un gruppo di *procedure* attinenti.
- Una gamma di *pettern parts*, utilizzati per costruire i corpi delle procedure e i valori nella memoria.

- Una finestra *program memory*, contenente rappresentazioni grafiche e/o testuali dei dati del programma.
- Una lista di *procedure definite*, che include sia procedure di sistema che definite dall'utente.
- Una collezione di *pulldown menu commands* per eseguire funzioni di editing del programma e di controllo.
- Zero o più *finestre di uscita ausiliarie*, che contengono le viste grafiche e/o testuali dei dati del programma.



Esternamente all'ambiente di sviluppo, c'è il robot e una pista su cui il robot opera. Quando i programmi sono definiti ed eseguiti, il loro scopo primario è dirigere le azioni del robot lungo la pista. Secondariamente, i programmi effettueranno i calcoli necessari per accertarsi che il robot effettui correttamente le relative mansioni.
Quando le procedure "vedono" cose, stanno osservando in due zone dell'ambiente di SeeDo:
- il robot e la relativa pista
- la memoria del programma

Quando un modello visuale contiene l'immagine del robot, la vista si sta facendo dalla prospettiva del robot.
Quando un modello non contiene il robot o elementi delle mattonelle della strada, la vista si sta facendo dal controllore di esecuzione del programma incorporato nell'ambiente di SeeDo. Quindi, un modello **see/do** che non contiene il robot o elementi delle mattonelle della strada ha l'effetto di un costrutto condizionale di un convenzionale linguaggio di programmazione imperativo.
Poiché SeeDo fornisce caratteristiche paragonabili ai linguaggi di programmazione general-purpose, è possibile scrivere programmi che non guidano il robot.
Nell'implementazione corrente del linguaggio SeeDo, tale programmazione "senza robot" non è di interesse immediato. Tuttavia, in futuro i progettisti del linguaggio pensano di espandere i settori di applicazione di SeeDo, compreso la programmazione visuale general-purpose.

Principi fondamentali Di Programmazione
Le finestre per editare programmi di SeeDo contengono le collezioni di *procedure*. Queste procedure sono paragonabili alle procedure e alle funzioni scritte nei linguaggi di programmazione convenzionali, con alcune caratteristiche aggiunte in termini di come possono essere invocate. Queste caratteristiche aggiunte sono descritte qui sotto.

Il formato generale di una definizione di procedura è il seguente:

to *procedure-name* (*optional-parameters*)
see/do pattern { [**and** | **or**] **see/do** pattern ...}
|
see/say pattern { [**and** | **or**] **see/say** pattern ...}
|
do pattern { [**and** | **or**] **do** pattern ... }
end

Prendendo in prestito la sintassi dal linguaggio di programmazione Logo, una procedura è definita con la parola chiave "**to**". Il corpo di una procedura contiene una o più forme d'azione di base:

- **see/do**
- **see/say**
- **do**

Una coppia **see/do** è intesa per essere l'azione primaria nel corpo di una procedura.
Un modello **see** si compone di elementi visuali e/o testuali che descrivono una delle cose seguenti:

- una configurazione del robot sulla strada
- il valore di un elemento di memoria di programma
- qualsiasi forma testuale di programma che computa un valore di verità

Un modello **do** si compone di elementi visuali e/o testuali che descrivono:

- una configurazione del robot sulla strada
- un'espressione che computa un valore

Un'azione **see/do** è una forma condizionale, paragonabile all'istruzione if-then in un linguaggio di programmazione convenzionale. Ci sono due differenze fondamentali fra un'azione **see/do** e un costrutto condizionale convenzionale:

a. Il modello in una clausola **see** può essere un'espressione booleana, o può essere un'immagine di qualcosa nell'ambiente di esecuzione. Più tipicamente, un modello **see** rappresenta un modello dove il robot è sulla strada.
b. Nel contesto di un programma completo, i modelli **see** costituiscono le basi per invocare procedure dirette ai modelli. Più propriamente, una procedura è invocata se il suo modello **see** è comparato con successo.

La seconda forma di procedura-azione è **see/say**. Questa è simile a **see/do**, eccetto che l'azione **say** significa che la procedura deve ritornare il valore dell'espressione che segue il **say**. Quindi, mentre un **do** causa che un'azione esterna è eseguita nell'ambiente del robot, un say causa che una procedura ritorna un valore al chiamante.
La terza forma di procedura-azione è **do**. Quando un'azione ha bisogno di essere eseguita incondizionatamente, è messa dopo un **do**. **Do** è usato tipicamente:

- per definire l'azione di default quando nessun modello **see** è comparato
- per chiamare esplicitamente procedure dall'interno delle azioni **do** di altre procedure
- per comandare il robot incondizionatamente, come in un comando di cambio di velocità
- mettere ed accedere a valori in memoria

Come già accennato, SeeDo è un linguaggio ibrido. In termini di definizione di procedure, questo significa che una procedura può essere invocata quando il suo modello **see** è comparato. Alternativamente, può essere invocata esplicitamente mediante il suo nome nell'azione **do** di un'altra procedura. In questo modo, SeeDo supporta sia chiamate a procedure standard che dirette ai modelli.
Una descrizione più dettagliata del linguaggio applicato alla risoluzione della sfida può essere trovata in: www.csc.calpoly.edu/~gfischer/vpc/AT-LARGE-SUBMISSION.html

Linguaggi di tipo Equation-Based

Forms/3

Form/3 è un linguaggio di programmazione visuale general-purpose.
Supporta astrazioni definite dall'utente, programmate attraverso la manipolazione diretta di un misto di grafici e testo in un ambiente idoneo, e incorpora esempi concreti e un minimo di concetti.
Form/3 è orientato verso la tecnologia object-oriented poiché da una molta enfasi all'astrazione sui dati.
Tale linguaggio incorpora un numero di tecniche originariamente concepite per spreadsheets.
In Form/3, non ci sono variabili e gli oggetti sono contenuti in *celle*. Un programmatore specifica una *formula* che computa l'oggetto che è contenuto nella cella.
Le celle sono organizzate in un gruppo chiamato *form*.
Una differenza tra Form/3 e gli spreadsheets è che il programmatore può dare una struttura a un programma organizzando le celle nelle form. Altre caratteristiche di Form/3 non comprese negli spreadsheets sono il supporto per l'astrazione sui dati, l'astrazione procedurale ed eventi di alto livello.
In Form/3, dati (valori) e operazioni (formule) sono ben intrecciati.
Ogni oggetto è contenuto in una cella ed è dichiarato nei termini di una formula. Le formule possono contenere riferimenti multipli a celle. Un oggetto può esistere solo attraverso la formula che lo definisce e ogni formula ha per risultato un oggetto.
Non c'è un esplicito passaggio di messaggi. Un'operazione è espressa come una formula e operazioni multiple sono definite separatamente in formule multiple, ognuna delle quali ha per risultato un nuovo oggetto.
Il programmatore di Form/3 inizia un programma creando una nuova form (selezionando da un menu del linguaggio le opzioni di manipolazione), assegnandogli un nome, instanziando delle celle create attraverso un menu nella form e definendo le formule.
La presenza di testo nelle formule di Form/3 è una caratteristica del linguaggio. Form/3 non mira a eliminare il testo completamente. Il suo obiettivo è di combinare le tecniche visuali (come la manipolazione diretta e il ritorno visuale continuo) con una flessibile mistura di testo e grafici, per aumentare la concretezza della programmazione. [10]

8. Confronto fra i Linguaggi Visuali per la Robotica

Si rappresentano in questa tabella i vantaggi e gli svantaggi di ogni tipo di approccio.

TIPO DI APPROCCIO	VANTAGGI	SVANTAGGI
Rule-Based (es. Altaira e Cocoa)	• Riducono gli ostacoli pedagogici • Sono particolarmente efficaci per far fronte al controllo di eventi reattivi	• Numero eccessivo di regole • Possibilità di conflitti tra regole • Non è chiaro come possano trattare strutture dati sofisticate • Difficoltà di astrazione per dati e procedure senza complicare in modo eccessivo l'applicazione delle regole • Anche quando le regole individuali sono facili da comprendere, ottenere una comprensione globale del programma nel suo insieme può essere difficile.
Dataflow (es. Prograph)	• Riducono degli ostacoli pedagogici • Supportano bene l'astrazione per dati e procedure	• Difficoltà nel far fronte al controllo degli eventi reattivi
Controlflow (es. VIPR)	• Sono particolarmente efficaci per far fronte al controllo di eventi reattivi • Trattano con sofisticate strutture dati	• Usano una rappresentazione testuale convenzionale per manipolare dati • Non è chiaro come possano ridurre gli ostacoli pedagogici
Equation-Based (es. Forms/3)	• Riducono degli ostacoli pedagogici • Supportano bene l'astrazione per dati e procedure	• Difficoltà nel far fronte al controllo degli eventi reattivi

9. Caso di Studio

Come conclusione del lavoro svolto, presentiamo un'idea per la soluzione del seguente problema. Il quesito nasce dall'articolo [16], in cui si propone un algoritmo per la navigazione autonoma di un

robot, basandosi sulla stima della posizione attraverso l'elaborazione ed il confronto di immagini e calcoli vettoriali.
Più precisamente, il metodo usato, il visual homing, consiste di due fasi:

1. matching phase, in cui si confronta la snapshot memorizzata del luogo vicino all'obiettivo con la snapshot percepita al momento;
2. navigation phase in cui tramite la valutazione della differenza della posizione e delle dimensioni degli oggetti si va verso l'obiettivo.

Il nostro compito è di valutare quale tra i linguaggi visuali presentati possa meglio adattarsi ad una problematica di questo tipo.
Innanzi tutto siamo noi a stabilire i criteri di valutazione. A differenza del Challenge, dove soprattutto l'aspetto pedagogico e la facilità di utilizzo erano i principali parametri di valutazione, riteniamo che in questo ambito siano ben più importanti l'efficienza e la flessibilità del linguaggio.
Quindi, mentre i linguaggi Rule-based, quali Altaira e Cocoa, ben figuravano al Challenge per il loro aspetto fortemente pedagogico e per la loro semplicità di utilizzo, nella nostra valutazione sono i primi linguaggi ad essere scartati, anche perché un problema così complesso causerebbe una esplosione di regole difficili da gestire anche graficamente.
Così, tra gli altri linguaggi analizzati, il più adatto ci sembra essere il Vipr, con l'aggiunta delle tecniche di zooming e di fish-eyeing, che permettono una facile ed intuitiva gestione di programmi anche ampi. Inoltre, il Vipr è un linguaggio efficiente ed il suo codice è riutilizzabile come le librerie in C. Il tutto, ovviamente, nuoce alla semplicità di programmazione, difetto che, come suddetto, non rientra nei nostri parametri di valutazione. Riteniamo, cioè, che tale problema sia da proporre a persone competenti e quindi in grado di programmare a certi livelli e non ad un pubblico con conoscenze mediocri della materia, per vedere cosa riesce a fare. Lo scopo per noi è quello di arrivare al prodotto e non di vedere se anche la gente non competente riesce ad arrivarci.
Le specifiche del problema sono le seguenti: un robot viene inserito in un ambiente in cui ci sono ostacoli (porte, sedie, comunque oggetti fissi per semplicità) e c'è un obiettivo la cui posizione viene fornita.
Possiamo intuitivamente suddividere la problematica in due sottoparti:

- la prima deve occuparsi di far "imparare" il percorso al robot;
- la seconda di effettuare la navigazione dal punto di partenza all'obiettivo.

Più in dettaglio, "imparare" significa:

1. acquisire una mappa della zona visitata;
2. fissare un punto da cui partire con la prossima missione.

Proponiamo una soluzione plausibile:
1. Il robot viene posizionato in un punto P1 (in figura 1) e comincia a muoversi acquisendo dati, per poi tornare al punto di partenza. Il "dove" muoversi è un problema; la nostra idea è di partire seguendo la linea d'aria congiungente l'obiettivo (se il goal viene fornito attraverso coordinate) o con una direzione pseudocasuale se l'obiettivo è riconoscibile attraverso un confronto con una snapshot memorizzata a priori.
Stiamo considerando un ambiente con ostacoli, per cui il percorso difficilmente può essere una unica linea retta dalla partenza all'obiettivo. Il robot scatterà, quando necessario, nuove snapshot che gli devono consentire una navigazione autonoma all'interno del percorso appena scoperto. Il robot si muove in una direzione fino a quando incontra un ostacolo (condizione percepibile attraverso dei sensori), torna nella situazione di partenza e si muove con un angolo diverso, di poco diverso rispetto all'angolo di partenza. A questo punto può trovare un altro ostacolo, scatta una nuova snapshot e torna indietro fino a quando si è mosso in tutte le direzioni.
Ora abbiamo informazione del tipo "c'è un ostacolo dopo 30 cm in direzione 30° a sud-est" e così via. Una procedura deve valutare quale è il punto più vicino al goal. Per far ciò considera i vettori che hanno come estremi il punto di partenza e il punto in cui, a causa di un ostacolo, il robot ha fatto ritorno.

A questo punto, il robot fissa questo secondo punto P2 come punto di origine per la seconda ricerca.

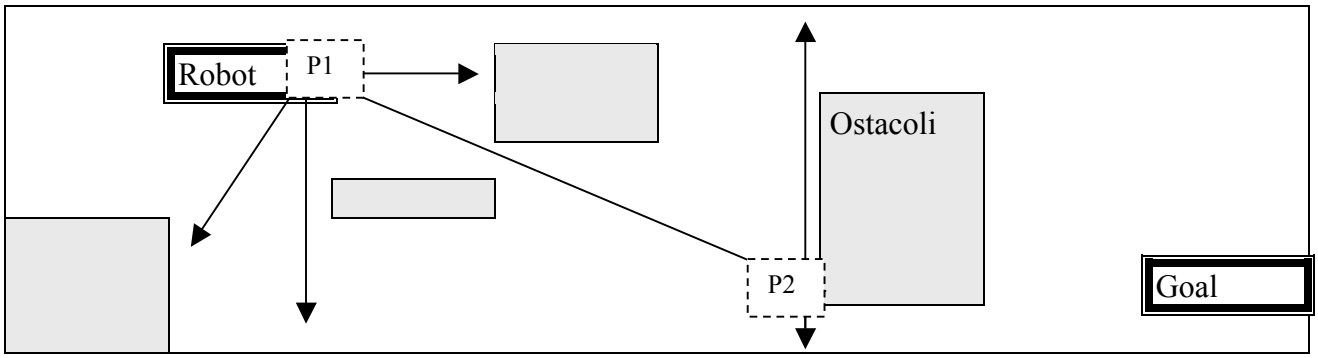


Figura 1: esempio di mappa di un ambiente con ostacoli.

La seconda parte deve occuparsi, invece, del "navigare"; navigare significa:

1. passare da un punto all'altro;
2. essere predisposto per cambiamenti della posizione del goal.

Una volta segmentato il percorso e fissati i punti, il robot si può spostare da un punto all'altro tramite un comando dall'esterno. Nel caso in cui venga cambiata la posizione del goal le conoscenze delle zone segmentate rimangono tali e semplicemente tramite il confronto delle snapshot memorizzate e di quella fornita dovrà ricercare all'interno di queste sottozone il nuovo obiettivo.
Le problematiche possono essere molte ed infatti vogliamo dare in questo lavoro solo un cenno di quali potrebbero essere le procedure necessarie a sviluppare un programma in Vipr per risolvere un tale problema.
Riassumiamo schematicamente il problema:

ROBOT:	Può muoversi in tutte le direzioni
	Può scattare snapshot
	Ha una zona di memoria per le snapshot
	Percepisce input forniti dall'utente
	Ha dei sensori per riconoscere ostacoli
AMBIENTE:	Costituito da oggetti vari
	Percorribile dal robot
	Ostacoli non mobili

A nostro parere servono le seguenti procedure:

- procedura di inizializzazione di tutte le variabili necessarie;
- procedura per il controllo dell'hardware (motori, telecamere, etc..); riguardo a questo punto il Vipr ci sembra il linguaggio più flessibile (tramite le estensioni Tcl);
- procedura per la gestione dell'area di memoria contenente le snapshot.
- procedura per l'algoritmo proposto in [16];
- procedura di acquisizione dati iniziali, quali la posizione del goal ed eventuali "markers" del percorso;
- procedura per decidere quando scattare le snapshot; secondo la nostra idea, si ha necessità di scattare una snapshot durante la fase di riconoscimento del percorso ogni volta che si incotra un ostacolo; successivamente, una volta noto il percorso e i segmenti, si può avere necessità di nuove snapshot per verificare se la rotta sia corretta e per aggiustarla nel caso non lo sia.

- Procedura per trovare i segmenti dalla partenza al goal; come ulteriore approfondimento potrebbe esserci la ricerca del percorso ottimo. Tale procedura deve colloquiare con una zona di memoria in cui si tiene traccia delle snapshot scattate, dei vettori e delle segmentazioni del percorso con i relativi punti di riferimento. L'idea già esposta in precedenza è quella di scattare le snapshot per ogni ostacolo trovato, quindi fare un confronto in base alla posizione del goal e quindi fissare il punto per una nuova ricerca, cancellando, se necessario, le snapshot precedentemente scattate (esclusa quella per la memorizzazione del punto iniziale del segmento);
- procedura che acquisisce comandi per spostarsi da un segmento all'altro. Infatti, possiamo ritenere l'obiettivo mobile e quindi fornire istruzioni al robot che autonomamente deve cambiare rotta;

Il problema è ovviamente molto complesso; tuttavia partendo con poche procedure e con molte semplificazioni, per poi aumentare il grado di complessità, riteniamo sia da ottimo appoggio un linguaggio visuale come il Vipr.

10. Bibliografia e Documentazione

[1] J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo & A. Repenning (1995)
LEGOSheets: A Rule-Based Programming, Simulation e Manipulation Environment for the LEGO Programmable Brick.

Proceedings of the 1995 IEEE Symposium on Visual Languages, Darmstadt,
IEEE Computer Society Press, pp. 172-179

[2] A.L. Ambler, T. Green, T.D. Kimura, A. Repenning & T.J. Smedley (1997)
1997 Visual Programming Challenge Summary
Proceedings of the 1997 IEEE Symposium on Visual Languages, Capri,
IEEE Computer Society Press, pp. 11-18

[3] **Special Section of Selected Papers from the 1997 Visual Programming Challenge**
Journal of Visual Languages and Computing (1998) **9**, *119-126*

[4] P.T. Cox, C.C. Risley and T.J. Smedly (1998)
Toward Concrete Representation in Visual Languages for Robot Control.
Journal of Visual Languages and Computing (1998) **9**, *211-239*

[5] P.T. Cox, T.J. Smedly, J. Garden, M. McManus
Experiences with Visual Programming in a Specific Domain
Visual Language Challenge '96
Proceedings of the 1997 IEEE Symposium on Visual Languages, Capri,
IEEE Computer Society Press, pp. 258-263

[6] P.T. Cox, T.J. Smedly
Visual Programming for Robot Control
Proceedings of the 1998 IEEE Symposium on Visual Languages, Halifax,
IEEE Computer Society Press, pp. 217-224

[7] Joseph J. Pfeiffer, Jr. (1998)
Altaira: A Rule-based Visual Language for Small Mobile Robots
Journal of Visual Languages and Computing (1998) **9**, *127-150*

[8] Joseph J. Pfeiffer, Jr. (1997)
A Rule-Based Visual Language for Small Mobile Robots
Proceedings of the 1997 IEEE Symposium on Visual Languages, Capri,
IEEE Computer Society Press, pp. 164-165

[9] Joseph J. Pfeiffer, Jr. (1998)
Case Study: Developing a Rule-Based Language for Mobile Robots
Proceedings of the 1998 IEEE Symposium on Visual Languages, Halifax,
IEEE Computer Society Press, pp. 204-209

[10] M. Burnett, A. Goldberg, T. Lewis
Visual Object-Oriented Programming
Concepts and Environments
MANNING, Greenwich

[11] P. Bottoni, M.F. Costabile, P. Mussio (1995)
Formalising Visual Languages
Proceedings of the 1995 IEEE Symposium on Visual Languages, Darmstadt,
IEEE Computer Society Press, pp. 45-52

[12] W. Citrin, S. Ghiasi, B. Zorn (1998)

VIPR and the Visual Programming Challenge
Journal of Visual Languages and Computing (1998) **9**, *241-258*

[13] W. Citrin, M. Doherty, B. Zorn (1998)
Formal Semantics of Control in a Complety Visual Programming Language
Proceedings of the 1994 IEEE Symposium on Visual Languages,
IEEE Computer Society Press, pp. 208-215

[14] W. Citrin (1996)
Incorporating Fisheyeing into a Visual Programming Environment
Proceedings of the 1996 IEEE Symposium on Visual Languages, Boulder,
IEEE Computer Society Press, pp. 20-27

[15] N. Heger, A. Cypher, D.C. Smith (1998)
Cocoa at the Visual Programming Challenge 1997
Journal of Visual Languages and Computing (1998) **9**, *151-169*

[16] A. Rizzi, G. Bianco, R. Cassinis (1998)
A Bee-inspired Visual Homing Using Color Images
Robotics and Autonomous Systems 25 (1998) 159-164

Altro materiale consultato:

[17] M.M. Burnett, M.J. Baker (1994)
A Classification System for Visual Programming Languages
Technical Report 93-60-14

[18] M. Erwig (1997)
Semantics of Visual Languages
Proceedings of the 1997 IEEE Symposium on Visual Languages, Capri,
IEEE Computer Society Press, pp. 304-311

[19] A. Repenning (1994)
Bending Icon: Syntactic and Semantic Trasformations of Icons
Proceedings of the 1994 IEEE Symposium on Visual Languages,
IEEE Computer Society Press, pp. 296-303

[20] H. Glaser, T.J. Smedly (1995)
The Next Generation of Command Line Interfaces
Proceedings of the 1995 IEEE Symposium on Visual Languages, Darmstadt,
IEEE Computer Society Press, pp. 29-36

[21] P.T. Cox, T.J. Smedly (1996)
A Visual Language for the Design of Structured Graphical Objects
Proceedings of the 1996 IEEE Symposium on Visual Languages, Boulder,
IEEE Computer Society Press, pp. 296-303

[22] Panel: **The 1994 Visual Languages Comparison**

Proceedings of the 1994 IEEE Symposium on Visual Languages,
IEEE Computer Society Press, pp. 90-97

[23] **Ten Years of Visual Languages Research**
Proceedings of the 1994 IEEE Symposium on Visual Languages,
IEEE Computer Society Press, pp. 196-205

[24] Panel: **Visual Languages and Programming in the Year 2004**
Proceedings of the 1994 IEEE Symposium on Visual Languages,
IEEE Computer Society Press, pp. 162-166

[25] R. Jamal, L. Wenzel
The Applicability of the Visual Programming Language LabVIEW to Large Real-World Applications
Proceedings of the 1995 IEEE Symposium on Visual Languages, Darmstadt,
IEEE Computer Society Press, pp. 99-106

11. Links

- www.cs.colorado.edu/~ralex/papers/
  è possibile reperire informazioni relative all'esperimento che ha dato lo stimolo per lanciare il 1997 Visual Programming Challenge
- www.designlab.unkans.edu/~ambler/index.html
  è possibile reperire informazioni relative al 1997 Visual Programming Challenge
- www.cs.dal.ca/~pcox/proj.html
  www.cs.dal.ca/~smedley/proj.html
  è possibile reperire informazioni relative all'applicazione di Prograph per risolvere il 1997 Visual Programming Challenge
- www.cs.nmsu.edu/text/Rsch/fac_projects.html
  hemicuda.cs.nmsu.edu/~altaira/submission
  www.cs.nmsu.edu/~pfeiffer/
  è possibile reperire informazioni relative all'applicazione di Altaira per risolvere il 1997 Visual Programming Challenge
- www.cs.orst.edu/~burnett/
  è possibile reperire informazioni relative a Form/3 e un'interessante bibliografia sui linguaggi visuali
- cocoa.apple.com/vpc/CocoaLego.html
  è possibile reperire informazioni relative all'applicazione di Cocoa per risolvere il 1997 Visual Programming Challenge
- www.csc.calpoly.edu/~gfischer/vpc/AT-LARGE-SUBMISSION.html
  è possibile reperire informazioni relative all'applicazione

di SeeDo per risolvere il 1997 Visual Programming Challenge

- www.cs.colorado.edu/~ghiasi/

  è possibile reperire informazioni relative all'applicazione di VIPR per risolvere il 1997 Visual Programming Challenge