UNIVERSITÀ CA' FOSCARI – VENEZIA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

Corso di Analisi e Verifica di Programmi

Marco Lionello: 810079

Analisi Statiche di programmi in JavaScript

Anno Accademico 2006-2007

Indice

Introduzione **5**

Principali caratteristiche di Javascript **7**

2.1 Javascript vs Java 7

2.2 Scrivere in JavaScript 8

2.3 Dichiarazione delle variabili 10

2.4 Document Object Model (DOM) 10

2.4.1 Oggetti 11

2.4.2 Proprietà 12

2.4.3 Metodi 12

2.4.4 Eventi 12

2.5 Errori in javascript 13

Proprietà dei programmi scritti in javascript **15**

3.1 Proprietà Generali 15

3.1.1 Proprietà verificabili tramite analisi dataflow 15

3.1.2 Compatibilità dei browser 16

3.1.3 Proprietà verificabili tramite interpretazione astratta 16

3.2 Proprietà Particolari 17

Dataflow analysis **19**

4.1 Dead code elimination 21

4.2 Constant propagation 21

4.3 Common subexpression elimination 22

4.4 Copy propagation 22

Analisi di incompatibilità dei browser **25**

5.1 Verifica di compatibilità dei browser 26

5.2 Analisi 26

5.2.1 Selezioni 26

5.2.2 Incompatibility(n) 27

5.2.3 Browser supportati(intraprocedurale, analisi forward data-flow, definite) 27

5.2.4 Contesto globale (analisi interprocedurale) 28

Interpretazione astratta **31**
6.1 Interpretazione astratta sul segno delle espressioni aritmetiche 33
Model checking analysis **37**
7.1 Form 38
7.1.1 Automa 38
7.1.2 Associazione di proprietà atomiche all'automa 40
7.1.3 Automa sulle stringhe valide 42
7.1.4 Associazione di proprietà atomiche all'automa 45
7.2 Automa generale 47
7.2.1 Proprietà 48
8 Conclusioni **51**
Glossario **53**
Bibliografia **55**

Indice delle figure

Figura 1: DOM 11
Figura 2: Esempio di interdipendenza 28
Figura 3: Automa 39
Figura 4: Automa con proprietà 41
Figura 5: Automa 43
Figura 6: Automa nome con contatore 44
Figura 7: Automa e-mail con contatore 44
Figura 8: Automa e-mail con proprietà 45
Figura 9: Porzione dell'automa generale 47

Capitolo 1

Introduzione

Questo progetto sviluppa alcune analisi di proprietà dei programmi scritti in JavaScript. Per prima cosa si analizzano le principali caratteristiche del linguaggio per poi individuare i possibili aspetti e le proprietà su cui si ritiene utile costruire un'analisi statica. Tali proprietà sono trattate in un primo momento a carattere generale, e quindi applicabili a gran parte dei programmi scritti in JavaScript, e successivamente a carattere particolare, e quindi applicabili solo a determinate classi di programmi. In ultimo si passa alla formalizzazione dell'analisi di tali proprietà.

Capitolo 2

Principali caratteristiche di Javascript

Javascript è un linguaggio di programmazione di pagine web relativamente semplice. Esso offre la possibilità di creare interattività addizionale alle pagine web. Con javascript è possibile trasformare pagine Web statiche in pagine che reagiscono e processano informazioni.

JavaScript nasce nel 1995 rilasciato da NetScape sotto il nome di LiveScript. Lo scopo principale di Netscape era quello di estendere le capacità dell'HTML statico per togliere lavoro ai server sovraccarichi e diluirlo nei computer locali.

Dopo l'introduzione di Java rilasciato da Sun Microsystems, LiveScript prese il nome di JavaScript. Microsoft riconoscendo l'importanza del nuovo linguaggio creò due linguaggi JScript e VBscript: il primo compatibile in parte con JavaScript, il secondo come sottoinsieme di Visual Basic.

Questi linguaggi, in competizione tra loro, hanno creato problemi agli sviluppatori fino a quando nel 1997 la European Computer Manufactores Association (ECMA) produsse la versione poi divenuta standard. Al giorno d'oggi non ancora tutti i programmi compilano su questo standard.

2.1 Javascript vs Java

Qui di seguito vengono riportate le principali differenze tra Javascript e Java.

Javascript	Java
Interpretato	Compilato
Object-Based	Object-Oriented
Non è stand alone. Richiede ed è embedded in HTML.	È stand alone. Java è un ambiente di sviluppo completo.
Sviluppato da Netscape.	Sviluppato da Sun Microsystems.
Debolmente tipato	Fortemente tipato

Alcuni esempi di programmi che possono essere scritti in JavaScript sono:

Rollovers	Validazione di form
Menù a cascata	Moduli e Bottoni
Online Quiz	Gallerie e Slideshow
Convertitori	Date e orari
Calcolatrici	Finestre Pop-Up

2.2 Scrivere in JavaScript

Come già accennato nei punti precedenti Javascript è un'estensione di HTML.

- Esso appare all'interno dei tag <HEAD>, <BODY>;
- Gli spazi bianchi in HTML non hanno importanza, lo hanno in Javascript.

Conoscere i modi in cui può essere inserito del codice Javascript è di fondamentale importanza per l'analisi del linguaggio. È possibile utilizzare varie tecniche; le tre principali sono:

- Lo <SCRIPT> e </SCRIPT> tag è la tecnica più utilizzata. È possibile inserire numerosi script tag nella stessa pagina HTML:

Tags	Cosa fa
`<SCRIPT LANGUAGE="JavaScript">`	Dice al browser che il codice JavaScript sta per apparire.
`</SCRIPT>`	Dice al browser che il codice JavaScript è finito.

- Un indirizzo URL JavaScript è uno speciale indirizzo URL che fa partire l'esecuzione di codice JavaScript:

URL JavaScript	Cosa fa
`<A HREF="JavaScript:void(0)">`	Dice al browser di non fare nulla quando si clicca sul link.
`<A HREF="JavaScript:window.history.go(-1)">`	Dice al browser quando il link viene cliccato, di accedere alla history e di caricare l'ultima pagina nella lista.

- L'event handlers esegue codice JavaScript per catturare gli eventi che occorrono nel browser; ad esempio, quando si muove il mouse sopra un link. Gli event handler cominciano sempre con il termine "on":

Event Handler	Evento attivatore	Cosa fa
onClick	Click del mouse sul link	Esegui il codice Javascript chiamato popUp(whichFlower).
onMouseOver	Puntatore mosso sopra il link	Crea una finestra pop-up e carica HTML o una JPG all'interno.
onMouseOut	Mouse mosso fuori dal link	Cancella la finestra pop-up

2.3 Dichiarazione delle variabili

I linguaggi tradizionali (come il C) utilizzano solo variabili che siano state precedentemente dichiarate per due motivi:

- ottimizzare l'uso della memoria;
- aumentare l'affidabilità.

Javascript, invece, utilizza un controllo di tipo lasco per cui non esiste, ne si rende necessaria una sezione di dichiarazione di variabili, dal momento che assegnato automaticamente il tipo in base alla dichiarazione. Ad esempio `Prova="testo"` e `Prova=59` sono due dichiarazioni pienamente valide, ma nel primo caso, Prova è considerata oggetto string, nel secondo la stessa è considerata come valore numerico.

Se valori diversi vengono concatenati prevale il valore string, per cui un numero concatenato ad una stringa produce una stringa, tuttavia se la stringa è un numero, il risultato sarà un numero (es.: `temval=365+"10"` darà `375`).

Tuttavia questo controllo non è sempre preciso (quando, ad esempio, si moltiplica una variabile stringa con un numero) e poichè non esiste un compilatore per controllare gli errori, lo script non genera i risultati voluti. In questo caso anzichè dichiarare, è sufficiente convertire il risultato con una funzione di conversione.

2.4 Document Object Model (DOM)

JavaScript è un linguaggio object-based. Gli oggetti sono i blocchi fondamentali. La maggior parte di quello che fa JavaScript interessa l'interazione di oggetti.

Gli oggetti possiedono:

- un insieme di caratteristiche (proprietà) che descrive come appariranno;

- un insieme di cose che possono fare (metodi) che ne descrivono il comportamento;
- un insieme di azioni che di cui sono in attesa, a cui rispondono (eventi).

In JavaScript il browser, tutte le immagini, i link, i bottoni, le form sono oggetti. Il DOM definisce e descrive questi oggetti, le loro proprietà e i metodi.

2.4.1 Oggetti

Gli oggetti interagiscono con il browser. Essi hanno una relazione gerarchica riportata nel diagramma seguente.

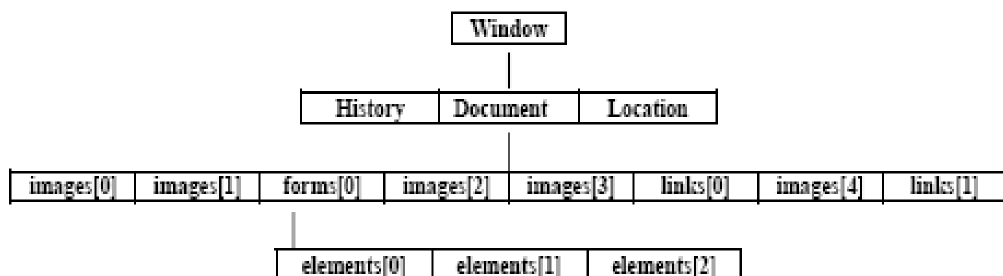

Figura 1: DOM

L'oggetto finestra è al livello più alto del DOM. Il documento è un sottoinsieme dell'oggetto finestra. JavaScript spezza l'oggetto documento in altri sottoinsiemi. Una piccola considerazione che può essere fatta sul nome degli oggetti è che può essere omesso nel path l'oggetto window: per esempio `window.document.images[0]` diventa `document.images[0]` . Se l'oggetto a cui intendiamo accedere risiede nella stessa finestra window, non è necessario specificare l'oggetto window, mentre se la finestra è diversa bisogna necessariamente specificarlo.

2.4.2 Proprietà

Le proprietà descrivono come un oggetto deve essere. Si possono cambiare le proprietà di un oggetto usando la dotNotation. Semplicemente si assegna un nuovo valore alla proprietà che si vuole modificare:

nome_oggetto.nome_proprietà = valore

2.4.3 Metodi

I metodi eseguono azioni su oggetti. Se vogliamo che appaia un pop-up con una immagine di fiori sarà necessario inserire il seguente codice che crea e gestisce il pop-up e richiamarlo opportunamente. Tutti i metodi utilizzati in questo codice sono in grassetto.

```
function Reader()
{
popup = window.open("","","width=100,height=125");
popup.document.open();
popup.focus();
po-
pup.document.write("<HTML><HEAD><TITLE>PopUp</TITLE><
/HEAD>
<BODY><img src='fiori.jpg'><BR>
<FORM><input type='button' value='Close' on-
Click='window.close()'></FORM>
</BODY></HTML>")
popup.document.close();
}
```

2.4.4 Eventi

Gli eventi sono azioni che occorrono nelle pagine Web. Con "Occorrono" intendiamo quando: il browser fa qualcosa, carica o scarica una pagina, si clicca su un bottone o

si muove il puntatore sopra un link. Questi eventi attivano dl codice JavaScript. Questo codice JavaScript viene chiamato event handler. Alcuni di essi sono:

Event Handler	**Evento**	**Oggetto**
`onClick`	Utente clicca su un oggetto	Button, Checkbox, Radio, Link, Reset, Submit, Area
`onChange`	Il valore di un oggetto(testo) cambia	Select, Text, Textarea
`onSelect`	Si selezione una text area	Text, Textarea
`onFocus`	Il mouse si sposta all'interno di un oggetto	Form Elements, Window
`onBlur`	Il mouse si muove fuori dall'oggetto	Form Elements,Window
`onMouseOver`	Il mouse si muove sopra dall'oggetto	Link, Area
`onMouseOut`	Il mouse si muove fuori dall'oggetto	Link, Area
`onSubmit`	A form è stata sottomessa.	Form
`onReset`	A form è resettata.	Form
`onLoad`	Un document o un'immagine ha finite di essere caricata.	Window
`onUnload`	Un documento è chiuso	Window
`onAbort`	Un utente termina il caricamento di un'immagine o di un documento	Image
`onError`	Un document o un'immagine non può essere caricata	Window, Image

2.5 Errori in javascript

Ci sono tre categorie di errori:

- *Load-time:* catturati da JavaScript quando il browser carica uno script. JavaScript mostra un warning box esplicitando il problema;

- *Run-time:* errore che occorre in tempo di esecuzione. JavaScript esplicita con una warning box la natura dell'errore;
- *Logic error:* quando lo script funziona ma dà risultati inaspettati.

Capitolo 3

Proprietà dei programmi scritti in javascript

In questo capitolo si analizzano le proprietà dei programmi scritti in JavaScript. Tali proprietà vengono suddivise in "Proprietà Generali" che possono essere applicate a qualsiasi programma e "Proprietà Particolari" che vengono applicate a determinati tipi di programma.

3.1 Proprietà Generali

Le proprietà che ricadono in questo gruppo sono le proprietà che possono essere applicate a qualsiasi programma scritto in JavaScript, indipendentemente da ciò che fa.

3.1.1 Proprietà verificabili tramite analisi dataflow

Le proprietà principali che i programmi scritti in JavaScript possono avere sono:

- l'assenza di comandi che assegnano un valore ad una variabile che non viene utilizzata prima di essere riassegnata, oppure l'assenza di comandi di assegnamento su variabili che non vengono mai utilizzate. Questa proprietà può essere verificata con un *dead code elimination* con una *liveness analysis*;

- l'assenza di variabili che sono usate come costanti. Per verificare questa proprietà si può usare il *costant folding* (o costant propagation) in una *reaching definitions* analysis;
- l'assenza di espressioni calcolate più di una volta. Per verificare questa proprietà si può usare una *common subexpression elimination* con una *available expression* analysis;
- l'assenza di due o più varibili che contengono sempre lo stesso valore. Per verificare questa proprietà si usa una *copy propagation* tramite una reaching definition analysis.

Le analisi di queste proprietà vengono formalizzate nel capitolo Dataflow Analysis.

3.1.2 Compatibilità dei browser

Come già visto nel capitolo precedente, JavaScript è un linguaggio per le pagine Web. Ogni applicazione scritta in JavaScript quindi non è stand alone, ma necessita di un ambiente di esecuzione. La prima proprietà che quindi risulta di fondamentale importanza è quella di verificare se un programma funziona con tutti gli ambienti (browser) senza lanciare eccezioni. Questa proprietà viene analizzata nel capitolo "Analisi di incompatibilità dei browser".

3.1.3 Proprietà verificabili tramite interpretazione astratta

Le proprietà dei programmi che sono verificabili tramite interpretazione astratta richiedono un'astrazione del dominio. Per quanto riguarda l'astrazione di un linguaggio con pochi costrutti di programma, si possono verificare proprietà molto interessanti, ad esempio, tra le più semplici si ricorda:

- proprietà sul segno delle espressioni aritmetiche;
- proprietà sul range di valori ammissibili da espressioni aritmetiche;

- proprietà sulla sicurezza;
- ecc.

Per quanto riguarda il linguaggio JavaScript, che è invece composto da molti costrutti, il discorso è differente. L'analisi di queste proprietà viene rimandata nel capitolo "Interpretazione astratta".

3.2 Proprietà Particolari

In questa sezione vengono proposte alcune proprietà che dipendono strettamente dall'esecuzione del programma.

Un esempio molto semplice di proprietà che dipendono dal programma è la necessità che in una form di immissione dati, di una pagina web scritta in JavaScript, tutti i dati siano corretti e siano presenti tutti i campi obbligatori.

Un esempio ulteriore può essere quello di garantire la proprietà che in una galleria di foto scritta in JavaScript, non si passi mai per la stessa foto.

Per verificare queste proprietà si utilizzeranno le tecniche di model checking, e per verificare proprietà logiche temporali si farà riferimento al CTL (Computation tree Logic).

Queste proprietà vengono specificate nel capitolo 7 Model Checking Analysis.

Capitolo 4

Dataflow analysis

In questo capitolo si formalizzano le proprietà descritte nella sezione "Proprietà Dataflow" del capitolo precedente.

Ricordiamo innanzitutto come sono definite le analisi dataflow. Per quanto riguarda la *liveness analysis* (la quale afferma che una variabile è live in un nodo *n* se contiene un valore che sarà utilizzato in futuro) abbiamo che:

- $gen_{LV}(p) = use[p]$
- $kill_{LV}(p) = def[n]$

$$LV_{exit}(p) = \begin{cases} \varnothing & \text{se p è un punto finale} \\ \bigcup \{ LV_{entry}(q) \mid \text{q segue p} \} & \end{cases}$$

$$LV_{entry}(p) = (LV_{exit}(p) \setminus kill_{LV}(p)) \cup gen_{LV}(p)$$

Def[n] è l'insieme di variabili definite nel nodo n, use[n] l'insieme di variabili usate nel nodo n, Lvexit(p) l'insieme delle variabili che sono *liveout* nel nodo p, Lventry(p) l'insieme di variabili che sono *livein* nel nodo p

La *reaching definition analysis* indica quali sono i comandi di assegnamento validi in un punto del programma:

$$RD_{entry}(p)= \begin{cases} \iota \text{ se p è il punto iniziale} \\ \cup \{ RD_{exit}(q) \mid \text{ q precede p}\} \end{cases}$$

$$RD_{exit}(p)= (RD_{entry}(p) \setminus kill_{RD}(p)) \cup gen_{RD}(p)$$

dove, p è un punto del programma, Rdentry(p) è l'insieme di coppie {(variabile,punto del programma),..} che sono definizioni valide nel punto p, Rdexit(p) le definizioni valide in uscita al punto p, gen(p) l'insieme di definizioni generate al punto p e kill(p) le definizioni uccise nel punto (p) e l�= {(x,?) | x è una variabile del programma}.

Infine, la *available expression analysis* ci dice quali espressioni sono state valutate e non successivamente modificate in un punto del programma. La formalizzazione è la seguente:

$$AE_{entry}(p)= \begin{cases} \emptyset \text{ se p è il punto iniziale} \\ \cap \{ AE_{exit}(q) \mid \text{ (q,p) è un arco del grafo}\} \end{cases}$$

$$AE_{exit}(p)= (AE_{entry}(p) \setminus kill_{AE}(p)) \cup gen_{AE}(p)$$

dove, p indica il punto del programma, kill(p) è l'insieme di espressioni uccise (per uccise si intendono le espressioni che contengono variabili modificate in quel punto), gen(p) sono le espressioni generate nel punto p (un'espressione generata si ha quando viene valutata in quel punto e nessuna variabile che vi compare viene modificata).

A questo punto possiamo andare a formalizzare le analisi delle proprietà ritenute utili in JavaScript.

4.1 Dead code elimination

La *dead code elimination* è un'analisi che elimina il codice inutile. La sua formalizzazione risulta molto semplice. Se c'è una istruzione del tipo s: a b c , tale che a non è live out di s, allora l'istruzione può essere eliminata.

La notazione *s: a b c* indica una istruzione in cui *s* è l'etichetta associata all'istruzione, *a* il primo registro in cui viene memorizzato il valore della computazione tra i registri *b* e *c*. La notazione *s: a b* indica una istruzione in cui al registro *a* viene copiato il valore contenuto nel registro *b*.

Alcune istruzioni hanno dei side effect impliciti. Per esempio, se il computer o il browser è impostato per lanciare un'eccezione quando si verifica un overflow o quando si divide per 0, allora l'eliminazione dell'istruzione causante l'eccezione può modificare il risultato della computazione.

L'ottimizzatore non dovrebbe creare modifiche che cambiano il comportamento del programma, sebbene il comportamento sembri benigno (ad esempio la rimozione di un errore a tempo di esecuzione). Il problema di queste ottimizzazioni consiste nel fatto che il programmatore non conosce mai il comportamento del programma e un programma debaggato con l'ottimizzatore può fallire quando l'ottimizzatore viene disabilitato.

4.2 Constant propagation

Supponiamo di avere una dichiarazione del tipo d: t c, dove c è una costante, e un'altra dichiarazione n che usa t, tale che n:y t x.

Noi sappiamo che t è costante in n se d raggiunge n, e nessun'altra definizione di t raggiunge n. In questo caso possiamo riscrivere n come y c x .

4.3 Common subexpression elimination

Dato una dichiarazione di un grafo di flusso s: t x y, dove l'espressione x y è disponibile in s, la computazione all'interno di s può essere eliminata.

L'algoritmo è il seguente:

1. computa la reaching definition per trovare le reaching expression (trova le dichiarazioni nella forma n: v x y , tali che il percorso da n a s non computa x y o definisce x o y);
2. scegli un temporaneo w, e per n riscrivilo come:
    - m: w x y;
    - n:v w;

3. infine modifica la dichiarazione s s: t w .

Bisogna affidarsi alla copy propagation per rimuovere gli assegnamenti in più.

4.4 Copy propagation

Questa è molto simile alla constant propagation, ma invece di avere una costante c abbiamo una variabile z. Supponiamo di avere una dichiarazione d:t z e un'altra dichiarazione n che usa t cioè n:y t x.

Se d raggiunge n, e nessun'altra definizione di t raggiunge n, e non c'è nessun'altra definizione di z in qualsiasi percorso da d a n (includendo i percorsi che attraversano n più volte), allora possiamo riscrivere n come y z x.

Se si esegue la copy propagation prima dell'allocazione nei registri, possiamo diminuire il numero di registri allocati. Ad ogni modo, questa analisi è utile anche per al-

tre ottimizzazioni, come ad esempio per la common-subexpression elimination. Per esempio, se si considera il seguente programma:

a y+z

u y

c u+z

le due espressioni di somma non vengono riconosciute come una subexpression elimination fino a che non viene efettuata una copy propagation di u a y .

Capitolo 5

Analisi di incompatibilità dei browser

Come si è evidenziato nell'introduzione, JavaScript è un linguaggio cruciale nello sviluppo del web. Purtroppo non esiste uno standard adottato al 100%. Inoltre JavaScript è stato implementato da varie organizzazioni in differenti versioni dei browser. Tipicamente la stessa interazione semantica (come ad esempio ottenere la larghezza della pagina correntemente visualizzata) è accessibile in vari modi. I programmatori si avvalgono delle strutture *control-flow* per selezionare dinamicamente il valore a cui accedere, ma questo dipende dall'ambiente in cui il codice opera. Queste tecniche usano una funzionalità di JavaScript che restituisce uno speciale identificatore quando si cerca di accedere a proprietà che non esistono. Quindi una chiamata a una proprietà o ad un metodo che non esiste lancia una eccezione del tipo NoSuchFieldExcepion (o NoSuchMethodException).

Ad esempio, la proprietà larghezza cambia in funzione del browser utilizzato, Internet Explorer 5 o 6, Mozilla (Firefox) e Safari. Quindi per ottenere la larghezza possiamo usare il seguente codice che è indipendente dal browser utilizzato:

```
function iw() {
  if (self.innerWidth) {
    return self.innerWidth;
  } else if (document.documentElement &&
           document.documentElement.clientWidth) {
    return document.documentElement.clientWidth;
  } else {
    return document.body.clientWidth;
  }}
```

Qui il primo “if” trova tutte le versioni di Internet Explorer (eccetto la versione 6). Il secondo “if” trova IE6, mentre l’ultimo “if” catttura tutti gli altri browser.

5.1 Verifica di compatibilità dei browser

Per prima cosa bisogna mettersi nelle condizioni di definire la struttura di un compilatore. Si può pensare che il parsing del codice dia come output un albero sintattico astratto grezzo (AST Abstract syntax tree).

Questo AST grezzo può essere trasformato in un albero concreto. Grazie a questa rappresentazione sarà possibile creare un grafo di controllo del flusso per ogni funzione (CFG Control flow graph). Il CFG per semplicità avrà una istruzione per blocco base.

A questo punto sarà possibile costruire una analisi dataflow, un costant folding e un dead-code elimination. Queste due ottimizzazioni possono ridurre la dimensione del CFG.

5.2 Analisi

In questa sezione si descrive l’analisi usata per la verifica della compatibilità dei browser.

5.2.1 Selezioni

Una selezione è un campo o un metodo richiesto in JavaScript. Le selezioni sono l’unità di base per la matrice di compatibilità del browser. Un esempio di selezioni è riportato qui di seguito:

- `self.screenX;`

- `document.documentElement["clientWidth"].`

5.2.2 Incompatibility(n)

Incompatibility(n) indica l'insieme dei browser incompatibili con la selezione n. Si può pensare a incompatibility come una matrice in cui le righe corrispondono alle selezioni. Una possibile formalizazzione è:

Incompatibility(n)={x / x ∊ {browser} e l'esecuzione di n in x lancia un'eccezione}

5.2.3 Browser supportati(intraprocedurale, analisi forward data-flow, definite)

Andiamo a questo punto a formalizzare l'insieme dei browser supportati. I parametri da inserire nel framework generale sono rispettivamente:

- E: punto iniziale
- l: {IE6, IE7, Safari, ecc.}={browser}
- F: nodi precedenti
- Operatore insemistico: $\cap$
- f: Lentry(n)\kill(n)

La formalizazzione diventa quindi:

$$\text{Gentry(n)}=\begin{cases}\{Browser\} & se\ n\ nodo\ iniziale \\ \cap\{Gexit(q) | q\ precede\ n\} & altrimenti\end{cases}$$

Gexit(n)=Gentry(n)\kill (n)

$$\text{Kill(n)}=\begin{cases}Incompatibility(n) & se\ n\ \text{è}\ una\ selezione \\ \emptyset & se\ n\ non\ \text{è}\ una\ selezione\end{cases}$$

A questo punto l'informazione contenuta in Gexit dell'ultimo nodo ci dà informazione riguardo ai browser supportati da una funzione p. Chiamiamo *supported(p)* tale informazione.

5.2.4 Contesto globale (analisi interprocedurale)

L'analisi finale considera l'intero programma e finalizza il risultato raccogliendo le precedenti analisi di contesto CFG. Se una funzione "funzione1" compatibile con {A,B,C} chiama un'altra funzione "funzione2" compatibile solo con {C} , allora funzione1 non è compatibile con {A,B}.

Esempio: prendiamo un programma che consiste di quattro funzioni m1, m2, m3, m4. Dopo l'analisi del contesto intraprocedurale del CFG, la funzione m1 è riportata compatibile con {A,B,C}, m2 compatibile con {A,B,C}, m3 compatibile con {B} e infine m4 compatibile con {A,B}.

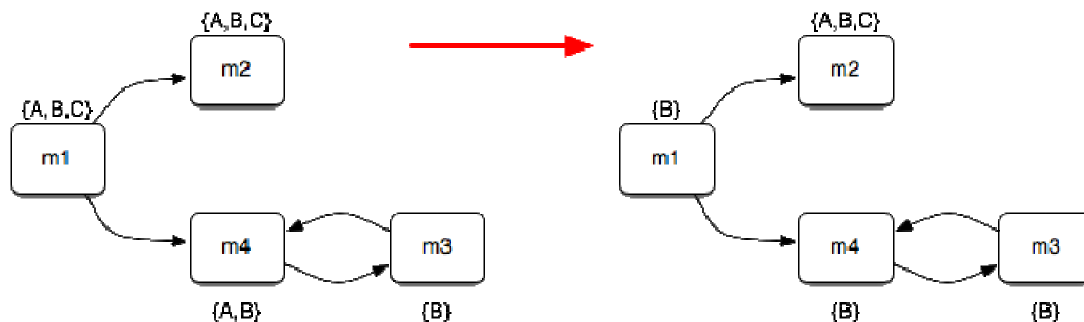

Figura 2: Esempio di interdipendenza

Dal momento che m4 chiama m3, esso è compatibile come m3 quindi solo compatibile con {B}. Questo risultato viene propagato a m1 in quanto è il chiamante di m4.

L'implementazione di questa analisi è elegante. Si modella un grafo di chiamate (call graph) come un CFG e si riusa la stessa analisi su questo modello. Attraverso il popolamento all'indietro del risultato, otteniamo l'analisi interprocedurale.

Formalizziamo quanto appena detto andando a definire i parametri del framework generale:

- E: punto finale
- l: {browser}
- F: successore
- Operatore insemistico: ⋂
- f: G1exit(n) ⋂ $supported(n)$

La formalizazzione diventa quindi:

$$G1exit(n)=\begin{cases} \{ Browser \} & se\ n\ nodo\ finale \\ \bigcap \{ G1entry(q) \mid q\ successore\ di\ n \} & altrimenti \end{cases}$$

G1entry(n)= G1exit(n) ⋂ $supported(n)$

Questa analisi è di tipo backward e definite. In G1entry del nodo iniziale si trovano i browser supportati dal programma.

Capitolo 6

Interpretazione astratta

Sviluppare una interpretazione astratta per programmi scritti in JavaScript risulta molto difficile. Infatti l'interpretazione astratta è una tecnica complessa, la quale:

- usa una semantica concreta, ovvero una funzione che assegna significati ai comandi di un programma in un dominio fissato;
- usa un dominio astratto che modella alcune proprietà della computazione concreta tralasciando le rimanenti informazioni;
- deriva una semantica astratta che permette di eseguire il programma sul dominio astratto per calcolare le proprietà che il dominio astratto modella;
- applica un algoritmo di punto fisso, il quale calcola staticamente una approssimazione corretta della semantica astratta.

L'interpretazione astratta inoltre:

- riduce l'interpretazione dei loop usando restringimenti sofisticati;
- potrebbe ritornare come restituire dei valori insignificanti (T);
- è molto complessa e costosa da implementare.

Alcune caratteristiche di JavaScript rendono l'interpretazione astratta, basata sull'analisi statica, ineffettiva in particolare:

- il carattere funzionale di JavaScript (in particolare l'abilità di ritornare un valore funzionale -implementato come chiusura- attraverso la funzione keyword), è difficile da analizzare attraverso l'interpretazione astratta;

- il carattere riflessivo di javascript, che include la primitiva *eval* non è trattabile dalle tecniche di interpretazione astratta. Questo è dovuto al fatto che implementare il reticolo astratto della funzione eval è molto difficile oppure rendendo più semplice il reticolo si avrebbe una perdita di informazioni in

quanto ritornerebbe sempre T. Alcuni esempi di utilizzo della funzione eval sono i seguenti:

```
<script type="text/javascript">
eval("x=10;y=20;document.write(x*y)")
document.write("<br />")
document.write(eval("2+2"))
document.write("<br />")
var x=10
document.write(eval(x+17))
document.write("<br />")
</script>
```

Che restituisce I valori 200, 4, 27 . Un altro esempio più complesso è:

```
p = new Object();
p[0] = "_df"; p[1] = "_ov";
p[2] = "_ot"; p[3] = "_dn";
function g(id, act)
{ if(document.images)
document.images[id].src = eval( "m." + id + p[act]
+ ".src");
}
if (document.images) {
m = new Object();
m.s_df = new Image(); m.s_df.src =
"/images/common/edu.gif";
m.s_ov = new Image(); m.s_ov.src =
"/images/common/edu2.gif";
m.s1_df = new Image(); m.s1_df.src =
"/images/common/service.gif";
m.s1_ov = new Image(); m.s1_ov.src =
"/images/common/service2.gif";
// etc...
}
```

La funzione eval valuta e/o esegue una stringa di codice JavaScript che è contenuta nell'argomento codestring. Prima determina se l'argomento è una stringa valida facendo un parse sul codice java. Se il parse è corretto esegue il codice e ritorna il valore dell'ultimo statement, se c'è un valore; se invece c'è un'espressione, essa viene valutata e il valore viene restituito. Se il codestring è vuoto restituisce "indefinito". Il carattere riflessivo è dato dal fatto che eval valuta espressioni.

- Le sue abilità di metaprogrammazione non sono trattabili dalle tecniche di interpretazione astratta specialmente nei casi che questa metaprogrammazione dipende dall'ambiente in cui opera.

Se si pensa che nel 46% delle pagine web contenenti JavaScript si fa uso di metaprogrammazione, non implementare l'interpretazione astratta sulla metaprogrammazione vorrebbe dire scartare il 46% dei codici da analizzare.

- Il carattere prototipato ad oggetti di JavaScript è inusuale per la comunità che studia l'interpretazione astratta.

Ognuno di questi punti, per essere effettivamente usato, dovrebbe essere un significativo punto base di ricerca accademica (senza alcuna garanzia che esso funzioni).

6.1 Interpretazione astratta sul segno delle espressioni aritmetiche

Come introdotto all'inizio di questo capitolo alcuni ricercatori affermano che risulta difficile creare un'interpretazione astratta su JavaScript, in quanto il lavoro sarebbe troppo faticoso o i risultati prodotti sarebbero privi di informazione.

Cerchiamo di formalizzare questo concetto. Per prima cosa formalizziamo il comportamento della funzione *eval*. La sintassi è la seguente:

```
eval(" codestring ");
```

Il comportamento viene riportato qui di seguito:

- se codestring è vuoto allora eval torna "undefined" ;
- se codestring non è una stringa valida ritorna "undefined" ;
- se la stinga è valida allora esegue il codice contenuto nella striga e ritorna il valore dell'ultima dichiarazione, se c'è una espressione valuta l'espressione e ritorna il valore appena valutato.

Un esempio è il seguente:

```
eval("fred=999; wilma=777; document.write(fred + wilma);");
```

```
Output:
1776
```

A questo punto possiamo andare a formalizzare una interpretazione astratta che dia informazioni sul segno di espressioni aritmetiche in un linguaggio semplice formato

solo dagli operatori *,+,/,- e non debolmente tipato, e successivamente estendiamo questa interpretazione al linguaggio JavaScript. Per non essere troppo prolissi si limita il discorso alla sola moltiplicazione.

La *semantica concreta* della moltiplicazione si può definire tramite la funzione μ definita come:

e= i | e*e | e+e | ….

μ: Exp⇾ Int

μ(i)= i

μ(e1 * e2)= μ(e1) * μ(e2)

La *semantica astratta*, che si limita a considerare solo il segno delle espressioni, è la seguente:

η: Exp⇾ { +, 0, - }

$$\eta(e1)=\begin{cases} + \; se \; e1 > 0 \\ 0 \; se \; e1 = 0 \\ - \; se \; e1 < 0 \end{cases}$$

η (e1 * e2)= η (e1) & η (e2)

&	+	0	-
+	+	0	-
0	0	0	0
-	-	0	+

Proviamo ora ad estendere questo semplice esempio in JavaScript, per permettere in seguito di definire della funzione *eval* . Aggiungiamo prima di tutto "*undefined*" (visto nella funzione eval). Aggiungiamo poi anche i possibili valori che le variabili nell'espressione possono assumere, ossia qualsiasi stringa. Questo è consentito nel linguaggio dal fatto che JavaScript è debolmente tipato. Chiamiamo *String* l'insieme delle stringhe. La semantica concreta diventa:

e= i | e*e | eval(e)

μ: Exp ⋃ {Undef} ⇾ Int ⋃ {Undef}

$$\mu(ei)=\begin{cases} ei \; se \; ei \neq Undef \; \wedge \; ei \; \notin \; String \\ \quad Undef \; se \; ei = Undef \\ \quad Undef \; se \; ei \; \in String \end{cases}$$

$$\mu(e1 * e2)=\begin{cases} \mu(e1) * \; \mu(e2) se \; e1, e2 \; \neq Undef \\ \quad Undef \; se \; e1 \mid e2 = Undef \end{cases}$$

quindi la semantica astratta diventa:

η: Exp + {Undef}→ { +, 0, -, ⊥, T }

$$\eta(e1)=\begin{cases} + \ se\ e1 > 0 \\ 0\ se\ e1 = 0 \\ -\ se\ e1 < 0 \\ Und\ se\ e1 = Undefined \\ String\ se\ e1\ \in String \end{cases}$$

η (e1 * e2)= η (e1) & η (e2)

&	+	0	-	Und	String
+	+	0	-	⊥	T
0	0	0	0	⊥	T
-	-	0	+	⊥	T
Und	⊥	⊥	⊥	⊥	T
String	T	T	T	T	T

A questo punto si può notare che solo nella la moltiplicazione abbiamo la presenza dell'elemento Top (T) e dell'elemento Bottom (⊥) e che quindi anche solo con l'operatore di moltiplicazione abbiamo perdita di informazione.

Proviamo a questo punto a definire la semantica concreta della funzione *eval*:

$$\mu_{eval}(e1, .., en)=\begin{cases} Undefinded\ se\ e1, \dots, en\ \notin Syntax \\ \mu(en)\ se\ en\ ha\ operatore * \ \wedge e1, \dots, en\ \in Syntax \end{cases}$$

dove *Syntax* è l'insieme delle stringhe valide (ossia le possibili stringhe di sintassi valide del JavaScript). La semantica astratta risulta:

η: Exp + {Undef}→ { +, 0, -, ⊥, T }

$$\eta_{eval}(e1,...en)=\begin{cases} \perp se\ e1, \dots, en\ \notin Syntax \\ \eta(en)\ se\ en\ ha\ operatore * \wedge e1, \dots, en\ \in Syntax \end{cases}$$

Si nota subito che l'interpretazione astratta del segno di una variabile in JavaScript, così come è stata definita, è molto limitativa infatti si può applicare solo con l'operatore di moltiplicazione. Per togliere questa limitazione è necessario valutare μ(en) in ogni sua forma, e quindi, definire, la semantica concreta completa del Java-

Script e creare la rispettiva semantica astratta. Ovviamente questo lavoro diventa paradossalmente complicato.

In alternativa si potrebbe pensare che la semantica concreta possa essere definita come:

$$\mu_{eval}(e1, .., en)=\begin{cases} \perp \; se \; e1, \dots, en \; \notin Syntax \\ \\ \quad \perp \; altrimenti \end{cases}$$

ma a questo punto si avrebbe una perdita totale di informazione e l'interpretazione astratta sarebbe inefficace.

ESEMPIO

Consideriamo la seguente scrittura:

```
5 * eval("fred=999; wilma=777; document.write(fred + wilma);");
```

- se prendiamo la prima semantica astratta di eval risulta che non è possibile applicare l'interpetazione astratta perchè document.write non è definito;
- per la seconda semantica di eval il risultato è $\perp$ (Perchè + & $\perp$).

Capitolo 7

Model checking analysis

Prima di andare nel dettaglio dei programmi scritti in JavaScript è utile ricordare cosa e come opera il model checking.

Il model checking è un metodo per verificare algoritmicamente i sistemi formali. Viene realizzato mediante la verifica del modello, spesso derivato dal modello hardware o software, soddisfacendo una specifica formale. La specifica è spesso scritta come formule logiche temporali.

Il modello solitamente viene espresso come un sistema di transizioni, cioè grafo orientato formato da nodi (o vertici) e archi. Un insieme di proposizioni atomiche è associato ad ogni nodo. I nodi rappresentano gli stati di un sistema, gli archi rappresentano le possibili esecuzioni che alterino lo stato, mentre le proposizioni atomiche rappresentano le proprietà fondamentali che caratterizzano un punto di esecuzione.

Formalmente il problema è posto così: scelta una proprietà da verificare, espressa come una formula logica temporale p, e un modello M avente stato iniziale s, decidere se $M, s \models p$.

Gli strumenti del model checking si scontrano con la crescita esponenziale dell'insieme degli stati, comunemente conosciuto come il problema dell'esplosione combinatoria, che serve a risolvere la maggior parte dei problemi del mondo reale.

Prima di cominciare è utile definire il modello che descrive il sistema. Una struttura di Kripke M su un insieme di proposizioni atomiche AP è una 4−upla M = (S, S0,R,L) dove:

- S è un insieme finito di stati;
- S0 ⊆ S è l'insieme degli stati iniziali;
- R ⊆ S×S è una relazione di transizione che deve essere totale, ossia per ogni stato s ∈ S esiste s1 ∈ S tale che sia definita R(s, s1);

• L : S → 2^{AP} è una funzione che assegna ad ogni stato un'etichetta che contiene le proposizioni atomiche vere in quello stato.

È possibile a questo punto tradurre le specifiche in termini di formule logiche utilizzando le logiche temporali CTL oppure LTL per verificare, utilizzando i model checker, che il modello soddisfi le proprietà, ossia che le proprietà risultino vere per tutte le possibili esecuzioni del modello.

7.1 Form

Come già visto nel capitolo 1, JavaScript ha innumerevoli funzionalità che gli permettono di essere un linguaggio di programmazione versatile a molte applicazioni sul web. Nella maggioranza dei casi però esso viene utilizzato per scopi ben precisi, riportati nella tabella 1.2 .

In particolare alcune statistiche indicano che nella maggior parte dei casi esso viene utilizzato per validare form di immissione dati in maniera tale da alleggerire il carico computazionale del server.

Andiamo quindi a cercare e verificare tramite tecniche di model checking alcune proprietà riguardanti i programmi che validano form. Cominciamo quindi a costruire l'automa relativo.

7.1.1 Automa

Si consideri una pagina web contenente 3 campi *Nome*, *Indirizzo* ed *e-mail*. I campi obbligatori sono rispettivamente Nome, e-mail. Non c'è ordine nell'immissione dei dati (è possibile riempire il campo e-mail prima del campo Nome ecc.). Inoltre in un primo momento possiamo pensare che non ci siano vincoli sui caratteri permessi e quindi l'immissione di un carattere in un qualsiasi campo produce l'evento. L'evento di immissione del nome viene indicato con N, l'evento di immissione dell'indirizzo con I, quello dell'e-mail con E ed infine il comando di validazione con T. L'automa (con unfolding) risulta essere il seguente:

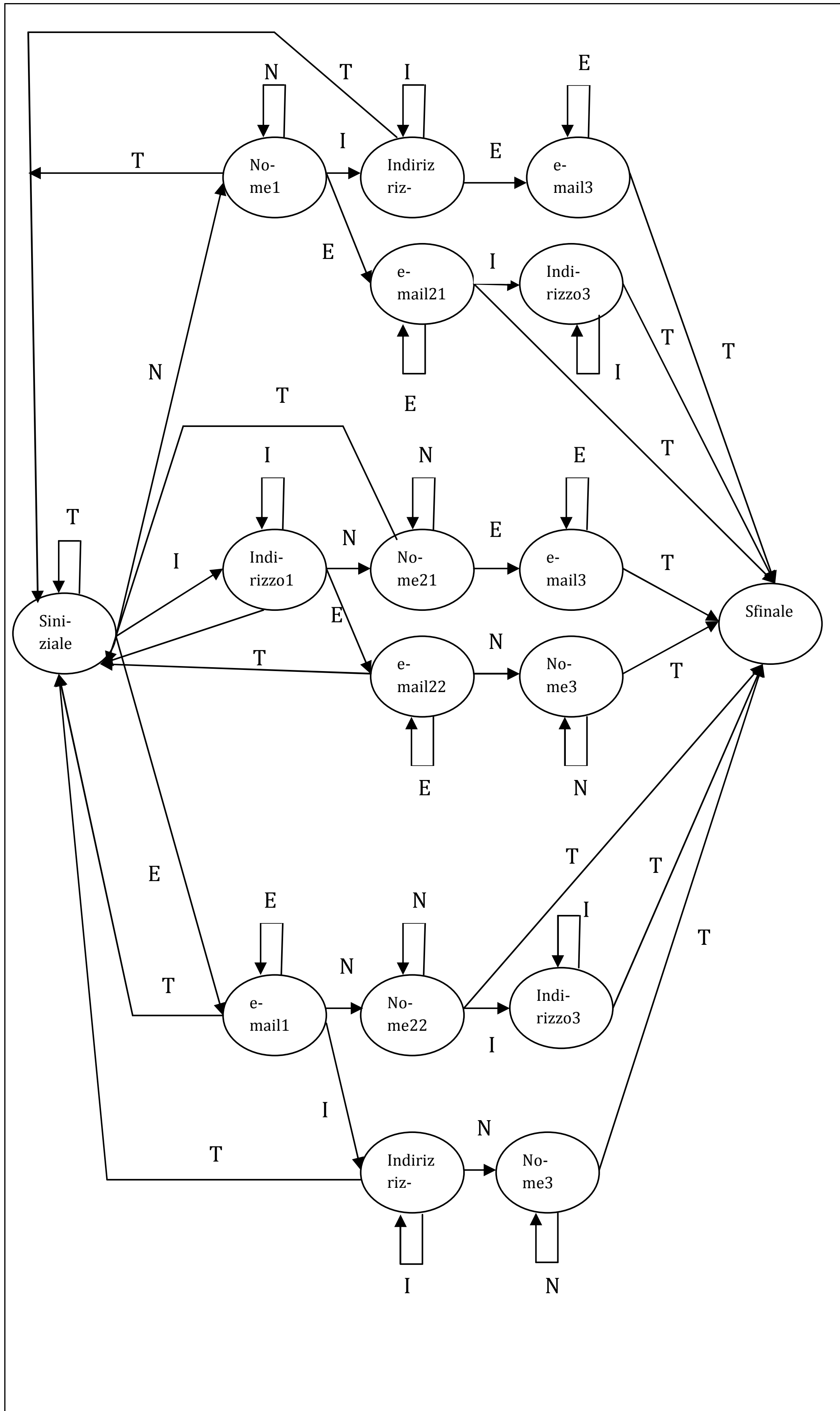

Figura 3: Automa

Lo stato iniziale rappresenta lo stato della pagina web nel momento in cui viene caricata. L'automa sopra riportato, quando riceve una richiesta di validazione della form, si comporta nelle seguente maniera:

- se i campi obbligatori sono presenti passa allo stato finale;
- se i campi obbligatori non sono presenti azzera le form e torna allo stato iniziale.

7.1.2 Associazione di proprietà atomiche all'automa

A questo punto possiamo associare all'automa alcune proprietà elementari (o atomiche) che sappiamo essere soddisfatte in quello stato.

La prima proprietà elementare che si può associare a tale automa è "la form è valida" che vale solo nello stato *Sfinale.* In tutti gli altri stati ovviamente la form non è valida. Inoltre allo stato iniziale possiamo associare la proprietà " la form è vuota".

Una seconda proprietà elementare che si può associare all'automa è che la form è valida solo se sono state riempite almeno le form Nome ed e-mail.

Andiamo ora a trasformare il nostro automa aggiungendo le propietà atomiche (o elementari) appena citate:

- Pn: è stata appena compilata la form Nome;
- Pi: è stata appena compilata la form Indirizzo;
- Pe: è stata appena compilata la form E-mail;
- Pok: la form è valida;
- Pv: le form sono vuote.

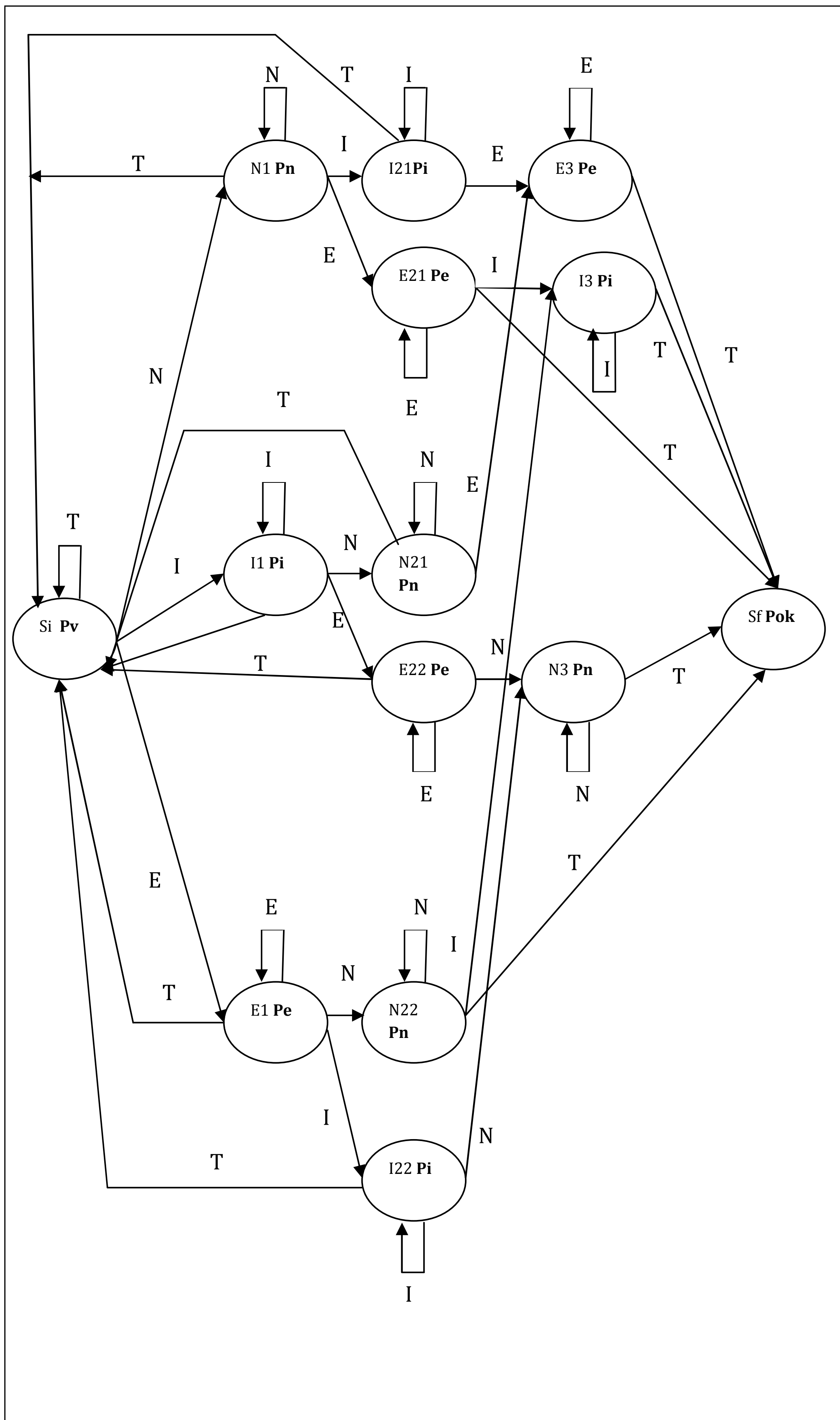

Figura 4: Automa con proprietà

La struttura di kripke è la seguente:

S = {N1 , I1 , E1 , N21 , N22 ,I21 , I22 , E21 , E22 , N3 , I3 , E3 , Si , Sf}

E = {N,I,E,T}

So = { Si }

L = {N1→{Pn}, I1→{Pi} , E1→{Pe} , N21→{ Pn } , N22→{ Pn } ,I21→{ Pi } , I22→{ Pi } , E21→{ Pe } , E22→{ Pe } , N3→{ Pn } , I3→{ Pi } , E3→{ Pe } , Si→{Pv} , Sf→{Pok} }

R= { (N1,N,N1) , (N1, I,I21), (N1, E , E21), (N1, T , Si), (I1,N,N21) , (I1, I, I1), (I1, E , E22), (I1, T , Si), (E1,N,N22) , (E1, I, I22), (E1, E , E1), (E1, T , Si), ………. }

Proviamo a vedere che ogni esecuzione che porta allo stato finale ha al suo interno almeno Pn e Pe. Questa proprietà vale e si può verificare con un model checker. La sua formalizzazione risulta essere:

Se Pok vale al tempo t:

allora $\exists\, t^{I} < t, t^{II} < t$*: Pn vale in* $t^{I} \wedge$ *Pn vale in* t^{II}

Una proprietà che invece non vale è la seguente: ogni esecuzione che porta allo stato finale contiene almeno Pn e Pi. Questa proprietà non vale e il model checker restituisce il contro esempio (Si → N1 → E21 → Sf).

7.1.3 Automa sulle stringhe valide

A questo punto andiamo a definire i tre automi che permettono di verificare che le stringhe inserite nelle form siano valide. Assumiamo che tutti e tre i campi possano avere una lunghezza massima di 30 caratteri. Nel nome non sono ammessi numeri,

nell'indirizzo invece sono permessi. Per quanto riguarda l'e-mail deve essere presente il carattere “@” in una posizione che non sia finale ne iniziale. Gli automi non tengono conto della possibilità di cancellare caratteri.

Per semplificare la comprensibilità dell'automa definiamo i seguenti simboli:

- L che rappresenta l'insieme {A , B,… , Z} ∪ $\{a , b , \ldots , z\}$;
- N che rappresenta l'insieme dei caratteri {0 , 1 , 2 , … , 9};
- A che rappresenta l'insieme dei rimanenti caratteri.

L'automa per il Nome è il seguente:

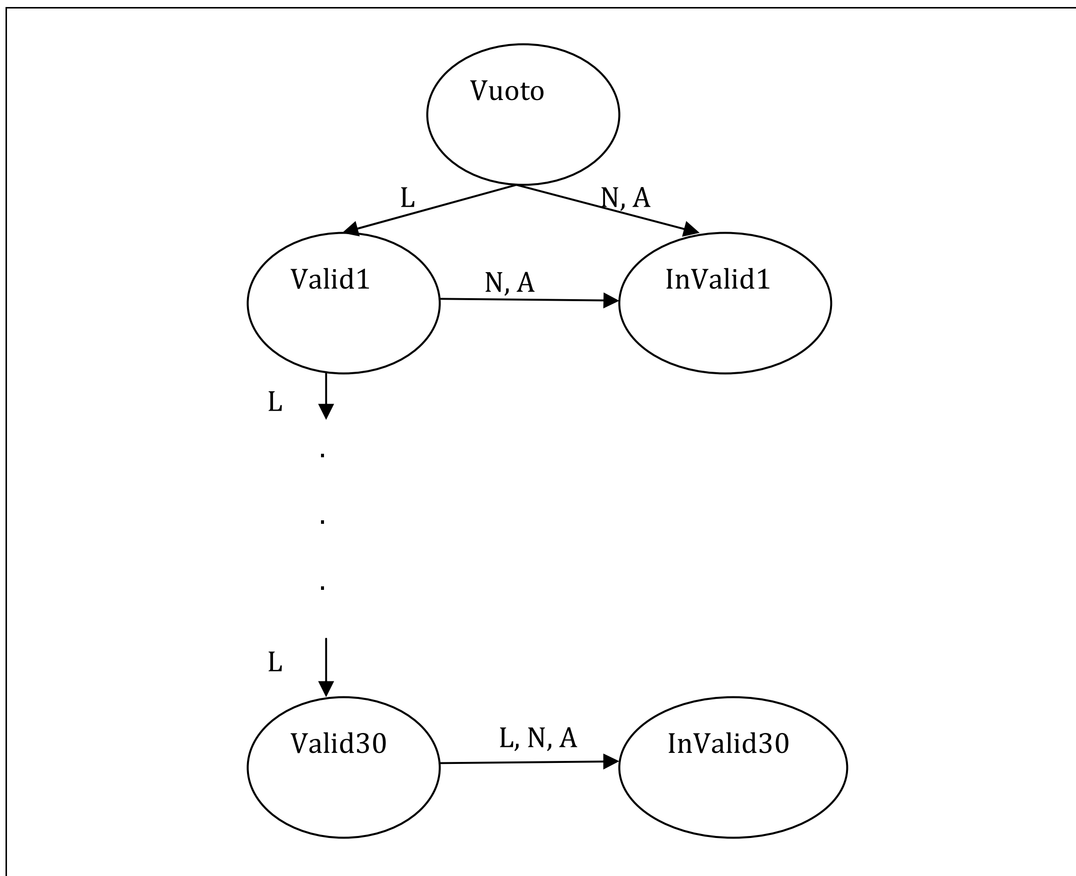

Figura 5: Automa

Un metodo alternativo di definire l'automa sopra riportato è quello di aggiungere una variabile che tenga conto del numero di caratteri inseriti. L'automa diventa quindi:

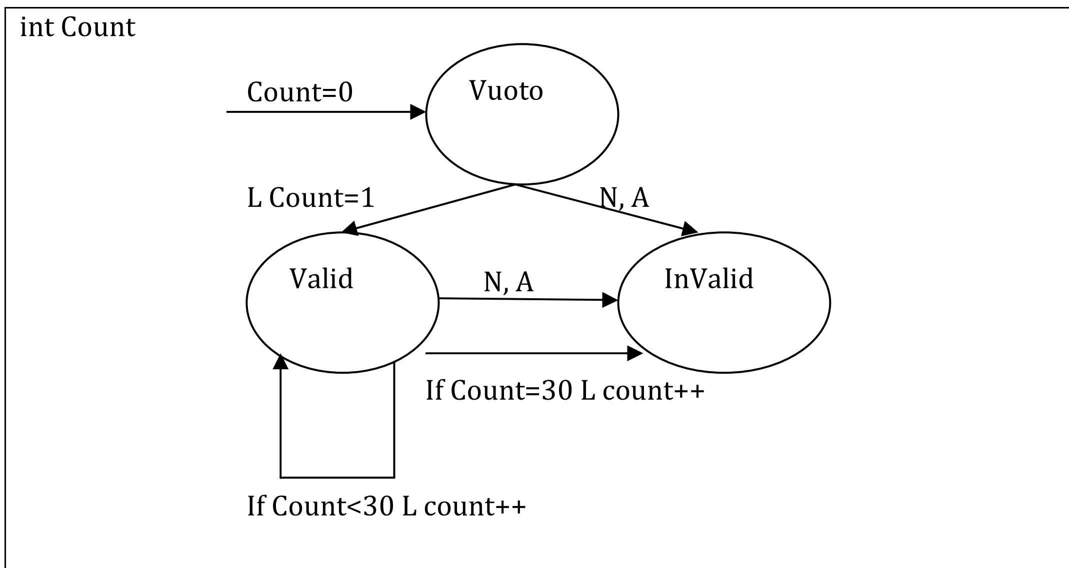

Figura 6: Automa nome con contatore

Per quanto riguarda la validazione dell'indirizzo l'automa è lo stesso, ad eccezione delle etichette negli archi. L'automa di verifica dell'e-mail è invece diverso e lo si riporta qui di seguito.

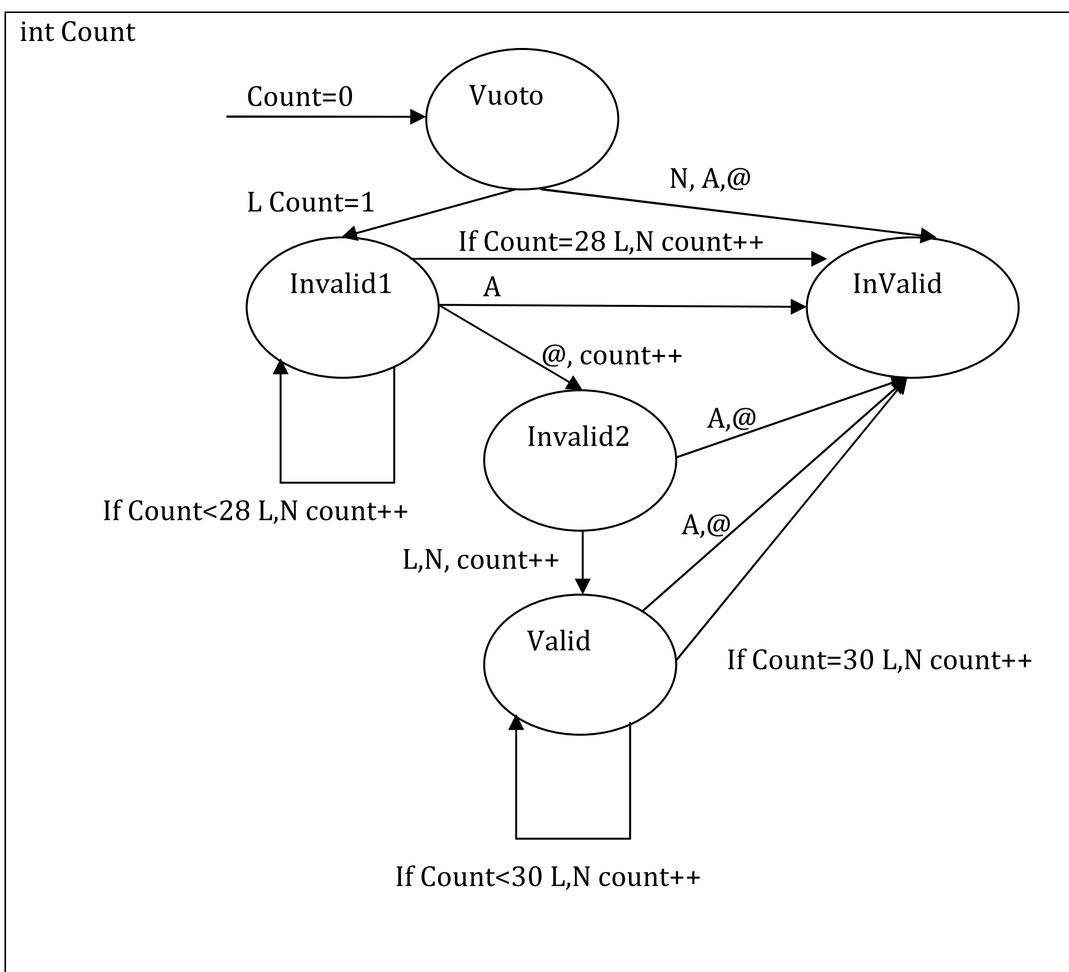

Figura 7: Automa e-mail con contatore

7.1.4 Associazione di proprietà atomiche all'automa

A questo punto possiamo associare all'automa alcune proprietà elementari (o atomiche) che sappiamo essere soddisfatte in quello stato nell'automa di inserzione dell'email.

La prima proprietà elementare che si può associare a tale automa è "l'e-mail è valida" che vale solo nello stato *Valid.* In tutti gli altri stati ovviamente l'email non è valida. Inoltre allo stato iniziale possiamo associare la proprietà " la form è vuota".

Una seconda proprietà elementare che si può associare nello stato *invalid2* è che è stato inserito il simbolo "@". L'automa aggiornato delle due proprietà precedenti risulta essere:

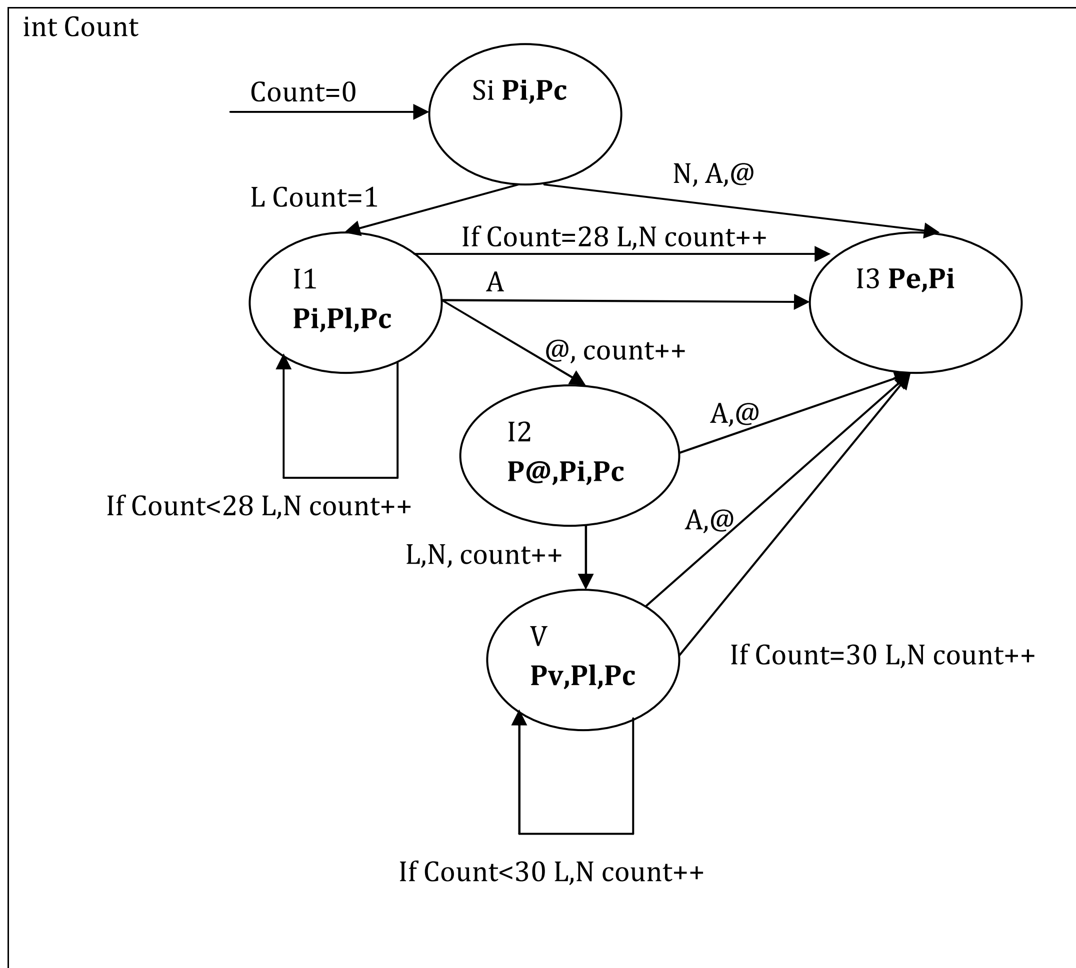

Figura 8: Automa e-mail con proprietà

dove:

- *Pv* è la proprietà "la stinga è valida";
- *Pi* è la proprietà "la stringa non è valida"*;*
- *P@* è la proprietà "è appena stata digitato @";
- *Pl* è la proprietà "è stato appena digitato L o N*";*
- *Pc* è la proprietà "il contatore è valido";
- *Pe* è la proprietà "è stato digitato un carattere non valido o count invalido".

La struttura di kripke è la seguente:

- S = { Si , V , I1 , I2 , I3};
- E = {N , A , L , @};
- So = { Si };
- L = { Si→{Pi}, V→{Pv} , I1→{Pi} , I2→{P@,Pi} , I3→{Pi} };
- R= { (Si,L,I1) , (Si,N,I3) , (Si,A,I3) , (I1,L,I1 if count<28) , (I1,L,I3 if count=28) , (I1,N,I1 if count<28) , (I1,N,I3 if count=28) , (I1,A,I3) , (I1,@,I2), (I2,@,I3) , (I2,A,I3), (I2,L,V) , (I2,N,V) , (V,L,V if count<30) , (V,L,I3 if count=30) , (V,N,V if count<30) , (V,N,I3 if count=30) , (V,A,I3)};

A questo punto possiamo andare a verificare con un model checker alcune proprietà, ad esempio:

- se la stringa è valida allora la stringa contiene una digitazione di L o N seguita da una del carattere @ e infine una digitazione di L o N. La sua formalizzazione è la seguente:

$$\forall t, \forall n \text{ se } vale\ Pv\ al\ tempo\ t\ allora:$$

$$\exists\, t^{I}, t^{II},\ /\ t^{I} < t^{II} \wedge\ t^{II}\ <\ t \wedge Pl\ vale\ in\ t^{I}\ \wedge P@\ vale\ in\ t^{II} \wedge Pl\ vale\ in\ t$$

- se la stringa è valida allora il contatore in tutti gli stati precedenti è valido:

$$\forall t, \forall n \text{ se } vale\ Pc\ al\ tempo\ t\ allora:$$
$$\forall t^I \ | \ t^I < t\ Pc\ vale\ in\ t^I$$

7.2 Automa generale

Unendo i tre automi visti nei punti precedenti possiamo creare l'automa generale in cui ad esempio sostituiamo al nodo E1 l'automa visto al punto 7.1.4. Aggiungiamo a questo punto I nuovi archi a tale automa, per tenere conto degli eventi {N,I,E,T}. Un'idea di come risulta l'automa generale viene riportato qui di seguito.

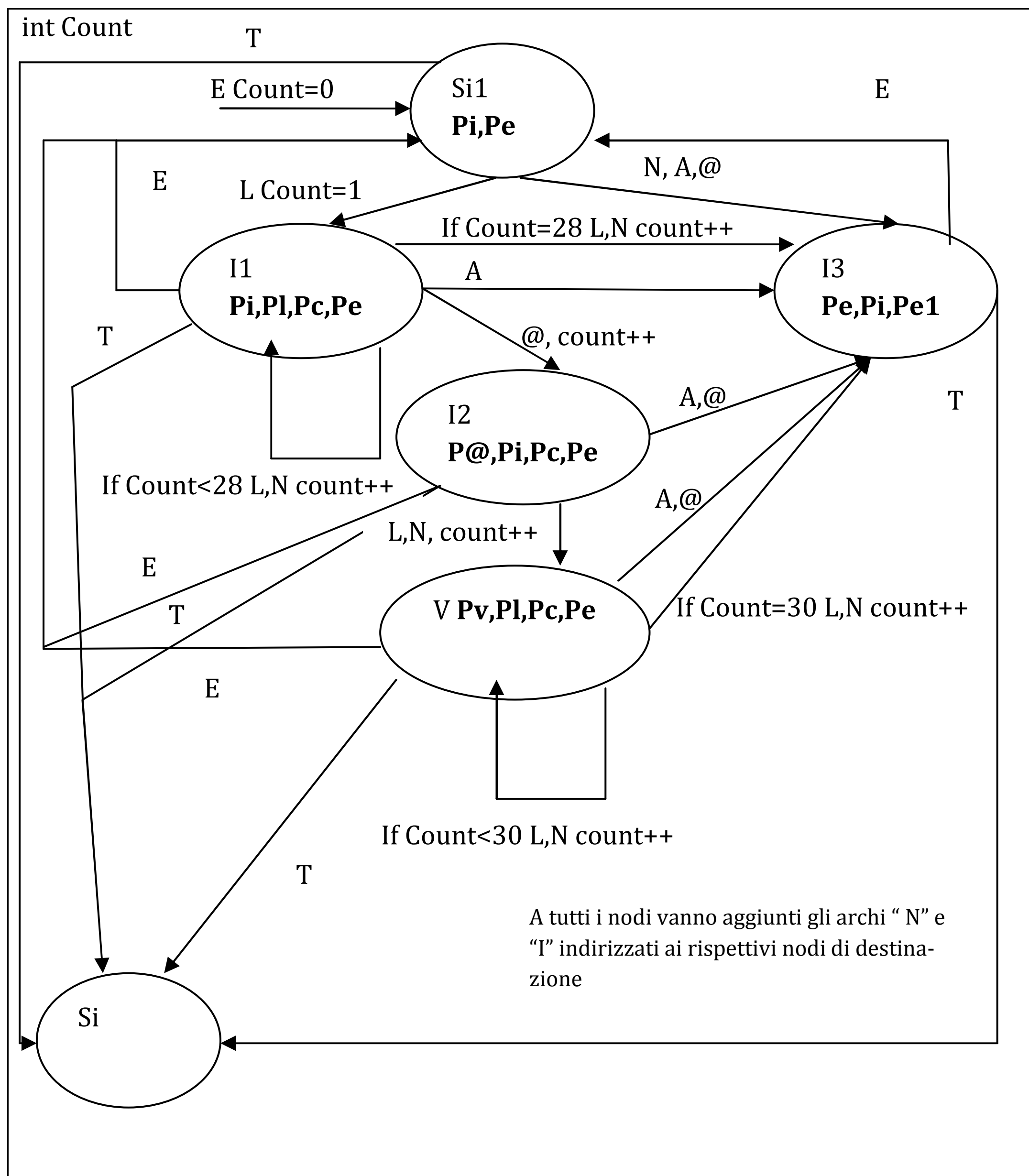

Figura 9: Porzione dell'automa generale

7.2.1 Proprietà

A questo punto è possibile definire delle proprietà sull'automa, che verifica se una form è corretta in base alle definizioni date ai punti precedenti. Andiamo a formalizzarle mediante regole temporali come LTL o CTL.

La prima categoria di proprietà che possiamo andare a verificare è quella di ***safety***, ossia proprietà che esprimono una condizione che deve sempre verificarsi oppure una condizione di pericolo che non si deve mai verificare per la sicurezza del sistema. Un esempio è il seguente:

$$CTL:\ AG\ \neg(Pok \wedge ((\neg Pn \wedge Pe) \vee (\neg Pe \wedge Pn) \vee (\neg Pe \wedge \neg Pn)))$$

che assicura che in ogni cammino sempre non può valere Pok se non sono presenti sia Pn sia Pe.

La seconda categoria di proprietà è quella di ***fairness,*** che è utile per descrivere il fatto che tutte le richieste devono prima o poi essere soddisfatte. Un esempio può essere:

$$CTL:\ AF\ \neg(Pi \wedge Pv)$$

che assicura che in ogni cammino prima o poi non vale Pi e Pv. In pratica descrive che sia Pi e Pv non possono valere contemporaneamente in nessun cammino dell'automa che verifica le stringhe.

La terza categoria di proprietà che si può andare a verificare è quella di ***liveness,*** che specifica qualcosa riguardo al futuro e talvolta è usata per identificare situazioni di errore. Un esempio può essere:

$$CTL:\ AG\ ((Pe \land Pn)\ U\ Pok\ )$$

8 Conclusioni

In questo progetto si sono descritte alcune proprietà dei programmi scritti in JavaScript. Dopo una panoramica sul linguaggio si sono divise le proprietà in "Generali" ossia quelle che riguardano tutti i programmi indipendentemente da ciò che essi fanno, e le "Particolari" ossia proprietà applicabili a determinati programmi. Queste proprietà sono state analizzate e formalizzate al fine di creare un'analisi statica che le verifichi.

Le tre analisi statiche che sono state formalizzate sono:

- Interpretazione Astratta
- Analisi Dataflow
- Model Checking

L'interpretazione astratta per come è definita risulta molto difficile da implementare e formalizzare in quanto il linguaggio JavaScript è molto complesso. L'analisi dataflow invece è molto versatile e si presta molto bene soprattutto se usata per verificare la proprietà di incompatibilità dei browser. Infine grazie al model checking si sono verificate alcune proprietà di programmi che sono largamente adottati utilizzando il linguaggio JavaScript.

Glossario

Linguaggio fortemente e debolmente tipato: Un sistema di tipi per un linguaggio di programmazione è un insieme di regole che consentono di dare un tipo ad espressioni, comandi ed altri costrutti del linguaggio. Un linguaggio si dice tipato se per esso è definito un tale sistema; altrimenti si dice non tipato. Il processo che porta alla determinazione di un tipo per i termini di un linguaggio si chiama controllo dei tipi (type checking).
Un linguaggio si dice fortemente tipato se il tipo di tutte le variabili è determinato a tempo di compilazione, altrimenti si dice dinamicamente tipato (se comunque esiste una nozione non triviale di tipo) o non tipato o debolmente tipato.
In un linguaggio fortemente tipato lo spazio di memoria richiesto per contenere il valore di ciascuna variabile durante l'esecuzione è completamente determinato a tempo di compilazione.

Programmazione funzionale: è un paradigma di programmazione in cui il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche. Solitamente questo approccio viene usato maggiormente in ambiti accademici piuttosto che industriali. Il punto di forza principale di questo paradigma è la mancanza di effetti collaterali (side-effect) delle funzioni, il che comporta una più facile verifica della correttezza e della mancanza di bug del programma e la possibilità di una maggiore ottimizzazione dello stesso.

Keyword di Javascript:

- la keyword this per riferirsi a qualunque oggetto istanziato dalla funzione costruttrice;
- la keyword prototype, per estendere le funzioni costruttrici e per ottenere una forma di ereditarietà di implementazione rispetto ad altre funzioni costruttrici;

Bibliografia

Compiler Design Implementation [Libro] / aut. Mucknick Steven. - [s.l.] : Morgan Kaufmann Publ, 1997.

Corso di Analisi e Verifica Programmi [Online] / aut. Tino Cortesi. - 01 08 2007. - http://www.dsi.unive.it/~avp.

Introduction to JavaScript [Rapporto] / aut. Fletcher Kathy. - [s.l.] : West Virginia University, 2002.

Model Checking Tutorial [Rapporto] / aut. Mazzoni Paolo. - [s.l.] : Politecnico di Milano, 2006.

Modern Compiler Implementation in Java [Libro] / aut. Appel A. W.. - [s.l.] : Cambridge Univ. Press, 1998.

Pascal-Louis Perez, Josh Wiseman CS343 [Online] / aut. Pascal-Louis Perez, Josh Wiseman. - Stanford University. - 11 10 2007. - http://cs343-spr0607.stanford.edu/index.php/Projects:Pascal-Louis_Perez,_Josh_Wiseman.

PICS and Javascript [Online] // www.poesia-filter.org/. - 09 09 2007. - http://www.poesia-filter.org/pdf/Deliverable_5_1.pdf .

Principles of Static Analysis [Libro] / aut. F.Nielson H.R.Nielson, C.Hankin. - [s.l.] : Springer Verlag, 1999.

Recipient Driven Correctness Framework for Mobile Code [Rivista] / aut. Padmanabhan Krishnan James Larkin, Phil Stocks. - Australia : Bond University.

Systems and Software Verifications [Libro] / aut. al. B.Berard et. - [s.l.] : Springer Verlag , 2001.

Tipi nei linguaggi di programmazione [Online] / aut. Pietro Cenciarelli. - 15 7 2007. - http://www.dsi.uniroma1.it/~lpara/DISPENSE/sezione_tipi.pdf.