Timed automata per applicazioni real-time

Tesina di

Metodi Formali nell'Ingegneria del Software

a.a. 2007-2008

SAPIENZA- Università di Roma

Autori:

Manuela Salvatori

Ivan Secci

Supervisore: prof. Toni Mancini

Indice

Capitolo 1: Sistemi real-time e Timed Automata

1.1 Sistemi real-time
1.2 Timed Automata
1.3 Sintassi dei Timed Automata
1.3.1 Definizione di Clock Constraints
1.3.2 Definizione di Timed Automaton
1.4 Semantica dei Timed Automata
1.4.1 Definizione di semantica di Timed Automata
1.5 Reti di timed Automata
1.5.1 Definizione di reti di timed Automata
1.6 Semantica di reti di timed automata
1.6.1 Definizione di semantica di una rete di timed Automata

Capitolo 2: Logiche temporali e model checking per i sistemi real-time

2.1 Computation Tree Logic (CTL)
2.1.1 Definizione di sintassi di CTL
2.1.2 Definizione di semantica di TCTL
2.2 Timed Computation Tree Logic (TCTL)
2.2.1 Definizione di sintassi di TCTL
2.2.2 Definizione di semantica di TCTL
2.3 Model Checking
2.4 Reachability Analysis

Capitolo 3: Panoramica su Uppaal e timed automata in Uppaal

3.1 Uppaal
3.2 Architettura di Uppaal
3.2.1 Linguaggio di modellazione
3.2.2 Simulatore
3.2.3 Verificatore
3.3 Timed Automata in Uppaal
3.4 Sintassi
3.4.1 Dichiarazioni di variabili locali e globali
3.4.1.1 Esempio di dichiarazione di variabile
3.4.2 Template dei processi
3.5 Semantica
3.6 Model Checking in UPPAAL

Capitolo 4: Applicazione pratica: rappresentazione e verifica del protocollo DHCP

4.1 DHCP (Dynamic Host Configuration Protocol)

4.2 Diagramma temporale

4.3 Diagramma degli stati e delle transizioni

4.4 Modellazione in UPPAAL

4.4.1 Dichiarazioni globali e sincronizzazione

4.4.2 Dichiarazioni e metodi locali

4.5 Simulazione del modello

4.6 Model checking

Bibliografia

Capitolo 1
Sistemi real-time e Timed Automata

1.1 Sistemi real-time

Nel corso dei decenni passati, la sfida dei ricercatori è stata lo sviluppo di teorie e tecniche che garantissero la correttezza dei sistemi. Quelle più spesso utilizzate, la simulazione e il testing, anche se efficaci, producono un eccessivo dispendio temporale e quindi economico, fornendo spesso solo stime statistiche di correttezza.

L'alternativa promettente consiste nell'applicare metodi rigorosi per provare la soddisfacibilità dei requisiti ad una descrizione formale del sistema. Tale descrizione consente di riconoscere errori prima ancora della fase implementativa, portando, come noto, un abbattimento dei costi sostenuti per risolverli. La tecnica che ha ulteriormente incoraggiato questa alternativa è noto con il nome di *model checking*: esso permette di analizzare le descrizioni dei sistemi dimostrando automaticamente se i requisiti dati sono soddisfatti generandone una prova.

Una classe particolare di sistemi cui si dedica la ricerca è quella dei sistemi real-time.

Un sistema real-time è un sistema la cui correttezza non solo dipende dai risultati, ma soprattutto dal tempo nel quale questi risultati sono prodotti.

E' possibile grossolanamente suddividere i sistemi real-time in due sottogruppi in base a quanto sia critico il mancato rispetto di specifici vincoli temporali:

- hard real-time: il tempo è fondamentale ed è necessario che il sistema risponda entro e non oltre certi limiti di tempo per non andare incontro a fallimento del sistema;
- soft real-time: non è detto che la violazione di una certa scadenza temporale determini un fallimento dell'intero sistema.

L'accezione di un sistema real-time racchiude diverse classi di sistemi che operano in ambiti differenti come quello ad esempio di embedded e safety-critical system, cioè un sistema grande e complesso composto da elementi eterogenei che lavorano in un ambiente safety-critical; oppure concurrent system, un sistema costituito da più componenti che lavorano in parallelo; o reactive system, ovvero un sistema real-time che deve rispondere prontamente a stimoli, segnali che riceve dall'ambiente esterno dove con "prontamente" si intende "in accordo con specifici vincoli temporali".

Il tempo gioca quindi un ruolo fondamentale ed è per questo che deve essere definito e mantenuto in maniera assai scrupolosa. Ogni sistema real-time è caratterizzato da un tempo globale di riferimento sia per il sistema sia per il suo ambiente che però non rappresenta necessariamente un clock globale. I

componenti di un sistema real-time, infatti, devono avere dei loro propri clock locali che possono essere testati e azzerati.

Per poter definire in maniera rigorosa un sistema real-time è necessario introdurre delle definizioni ausiliarie e fondamentali ai fini della definizione stessa di un sistema real-time.

1.2 Timed Automata

La nozione di Timed Automata fu sviluppata da Alur e Dill intorno agli anni 90 come un'estensione di automi a stati finiti con variabili di clock.
La ragione che li spinse a modellare questi automi nacque dal fatto che le tecniche tradizionali per il model checking non consentivano una modellazione esplicita del tempo e perciò non erano adatte ad analizzare i sistemi real-time la cui correttezza non dipende solamente dai risultati ma anche e soprattutto dal tempo nel quale i risultati vengono prodotti.
I timed automata quindi furono introdotti per consentire, in modo semplice, di collegare le transizioni sugli archi del grafo a vincoli di tempo utilizzando delle variabili di clock a valori reali.
Una versione semplificata dei timed automata, chiamati *Timed Safety Automata,* è nata per descrivere l'evoluzione del sistema mediante l'uso di condizioni locali invarianti. I Timed Safety Automata sono utilizzati in diversi tool di verifica, anche Uppaal li utilizza, e generalmente sono detti semplicemente Timed Automata.

1.3 Sintassi dei Timed Automata

Un timed automaton è un automa a stati finiti, cioè un grafo contenente un insieme finito di nodi o locazioni e un insieme finito di archi etichettati, esteso con un insieme di variabili reali, che può essere considerato come un modello astratto di un timed system.
Le variabili reali utilizzate nella rappresentazione del timed automaton consentono di modellare i clock logici del sistema; inizialmente vengono inizializzate a zero ed in seguito incrementate in modo sincrono con la stessa frequenza.
I vincoli di clock, per esempio le guardie sugli archi, sono utilizzati per restringere il comportamento dell'automa. Una transizione, rappresentata con un arco, può essere attraversata quando il valore del clock soddisfa la guardia che è posta sull'arco. Una volta che un arco è stato attraversato, i clock possono venire nuovamente inizializzati a zero.
Passiamo adesso ad una definizione rigorosa di clock e di timed automata.
Sia C un insieme finito di variabili a valori reali del tipo *x,y,ecc* rappresentanti i clock.
Sia $\sum$ un alfabeto di simboli del tipo *a,b,ecc* rappresentanti le azioni.

Definizione 1.3.1 (Clock Constraints) Un vincolo di clock (Clock Constraint) è una formula congiuntiva di vincoli atomici del tipo:

$x \sim n$ oppure $x - y \sim n$ per $x, y \in C$, $\sim \in \{\leq, <, =, \geq, >\}$ and $n \in N$.

Indicheremo con $B(C)$ l'insieme dei vincoli di clock.

Definizione 1.3.2 (Timed Automaton) Un Timed Automaton A è una tupla $\langle N, l_o, E, I \rangle$ dove

- N è un insieme finito di locazioni o nodi
- $l_o \in N$ è lo stato iniziale
- $E \subseteq N \times B(C) \times 2^C \times N$ è l'insieme di archi
- $I : N \rightarrow B(C)$ è una funzione che assegna gli invarianti alle locazioni

Scriveremo inoltre

$l \xrightarrow{g,a,r} l'$ quando $\langle l, g, a, r, l' \rangle \in E$ che rappresenta un arco dal nodo l al nodo l' con vincolo di clock g (anche detto condizione di attivazione o guardie dell'arco), un'azione a che deve essere eseguita e l'insieme di clock r da azzerare.

I(l) è la condizione invariante del nodo l che intuitivamente significa che tale vincolo di clock deve essere soddisfatto dai clock di sistema ogni qualvolta il sistema si trova nello stato l.

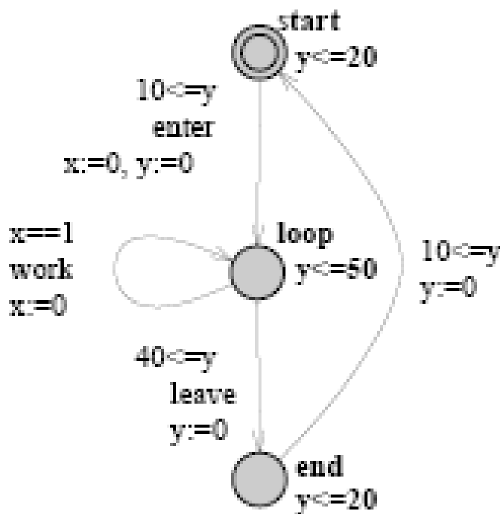


Figura 1: Timed Automaton

In figura 1 è rappresentato un semplice timed automaton nel quale possiamo riconoscere gli invarianti di ogni stato: *y<=20* per lo stato start, *y<=50* per lo stato *loop*, *y<=20* per lo stato *end*. Nell'arco che unisce *start* a *loop* si possono individuare la condizione di attivazione *10<=y*, l'azione *enter* e il reset di *x* e *y*.

1.4 Semantica dei timed Automata

La semantica di un timed automaton è definita come una transizione di sistema dove lo stato è costituito dalla locazione corrente e dal corrente valore del clock.

Per tenere traccia dei cambiamenti dei valori dei clock si usa una funzione chiamata *clock assignments* che mappa l'insieme C a valori reali non negativi R_+. Data una funzione di assegnazione u, scriviamo $u \in g$ per indicare che i valori di clock denotati da u soddisfano la guardia g. Per $d \in R_+$, $u + d$ indica l'assegnazione di clock che mappa *tutte* le $x \in C$ a $u(x) + d$ e per $r \subseteq C$, $[r \mapsto 0]u$ rappresenta l'assegnazione di clock che mappa tutti i clock in r a 0 e concorda con u per tutti gli altri clock in $C \setminus r$.
Con $u \in I(l)$ indichiamo invece che la funzione u soddisfa l'invariante sul nodo l.

Definizione 1.4.1 (Semantica) La semantica di un timed automata è una transizione di sistema (anche conosciuta come transizione di sistema temporizzata) dove gli stati sono coppie $\langle l,u \rangle$ dove l è un nodo e u un'assegnazione di clock, e le transizioni sono definite dalle seguenti regole:

- $\langle l,u \rangle \xrightarrow{d} \langle l,u+d \rangle if\ \ u \in I(l)\ and\ (u+d) \in I(l)\ \ per\ d \in R_+$ (*delay transition*)
- $\langle l,u \rangle \xrightarrow{a} \langle l',u' \rangle\ \ if\ \ l \xrightarrow{g,a,r} l', u \in g, u' = [r \mapsto 0]u\ \ and\ u' \in I(l')$ (*action transition*)

Come si può osservare, esistono due tipi di transizioni tra stati: l'automa può ritardare per qualche tempo la transizione (delay transition) oppure può attraversare un arco abilitato (action transition).

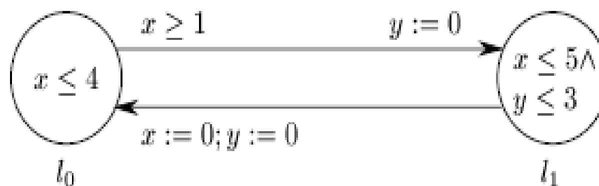


Figura 2

Nell'esempio in figura 2, lo stato l_0 ha il vincolo di clock $x \leq 4$ come invariante, mentre sull'arco da l_0 a l_1 è riportato il vincolo di clock $x \geq 1$ (guardia).
Nel momento in cui ad x viene assegnato il valore 1 il sistema può compiere due possibili transizioni:

- rimanere nello stato l_0 (delay transition);
- passare allo stato l_1 (action transition).

Quando ad x viene assegnato il valore 4 e lo stato attuale è ancora l_0, l'invariante forzerà la transizione che porta il sistema nello stato l_1.
y:=0 rappresenta il reset del secondo clock.

1.5 Rete di timed Automata

Definizione 1.5.1 (Rete di timed automata) Una rete di timed automata è una composizione parallela $A_1 | A_2 | ... | A_n$ di un insieme di n timed automata $A_1, A_2, ..., A_n$, dove $A_i = (N_i, l_i^o, E_i, I_i)$ con $1 \leq i \leq n$, chiamati processi, combinati all'interno di un unico sistema.

La comunicazione sincrona tra i processi è realizzata mediante hand-shake utilizzando azioni di input e di output.

Per modellare la comunicazione sincrona l'alfabeto dei simboli di azione $\sum$ è ampliato con simboli per rappresentare le azioni di input, denotate con ***a?***, le azioni di output denotate con ***a!*** e le azioni interne denotate con τ .

La comunicazione asincrona tra i processi è realizzata mediante variabili condivise.

1.6 Semantica di reti di timed automata

La semantica di una rete di timed automata è data in termini di transizioni di sistema, così come la semantica di un singolo timed automaton, dove lo stato della rete è una coppia $\langle l,u \rangle$ con $l = (l_1, l_2, ..., l_n)$ vettore di locazioni correnti nella rete, un elemento per ogni processo, ed u funzione di clock assignment che mantiene il valore corrente dei clock nel sistema.

La rete di automi deve garantire due tipi di transizioni:

- delay transition dove l'invariante del vettore di locazioni è la congiunzione degli invarianti delle locazioni dei processi;
- discrete transition che si dividono in:
  - local action che i processi effettuano all'interno di loro;
  - synchronizing action che prevede che due processi si sincronizzino su un canale e si muovano simultaneamente.

Definizione 1.6.1 (Semantica di una rete di timed automata) La semantica di una rete di timed automata è una transizione di sistema (anche conosciuta come transizione di sistema temporizzata) dove gli stati sono coppie $\langle l,u \rangle$ sopra definite e le transizioni sono definite dalle seguenti regole:

$\langle l,u \rangle \xrightarrow{d} \langle l,u+d \rangle$ se $u \in I(l)$ e $(u+d) \in I(l)$ con $I(l) = \wedge_i I(l_i)$; (delay transition)

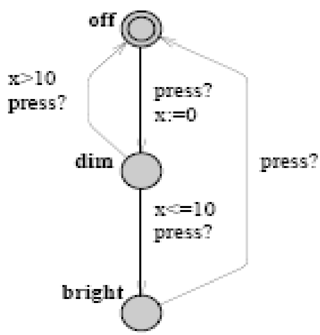
$\langle l,u \rangle \xrightarrow{\tau} \langle l[l'_i / l_i], u' \rangle$ se $l_i \xrightarrow{g,\tau,r} l'_i, u \in g, u' = [r \mapsto 0]u, u' \in I(l[l'_i / l_i])$ dove $l[l'_i / l_i]$ significa che l'i-esimo elemento l_i del vettore l è rimpiazzato dall'elemento l'_i ; (local action)

$\langle l,u \rangle \xrightarrow{\tau} \langle l[l'_i / l_i][l'_j / l_j], u' \rangle$ se esiste un indice $i \neq j$ tale che:

1. $l_i \xrightarrow{g_i, a?, r_i} l'_i, l_j \xrightarrow{g_j, a!, r_j} l'_j$ e $u \in g_i \wedge g_j$;
2. $u' = [r_i \cup r_j \mapsto 0]u$ e $u' \in I(l[l'_i / l_i][l'_j / l_j])$. (synchronizing action)

Esempio 1.6.2 (Rete di timed automata):

Consideriamo un sistema costituito dai due automi della figura 3a, il cui prodotto è dato dall'automa in figura 3b.



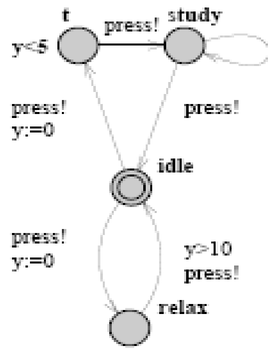


Figura 3a

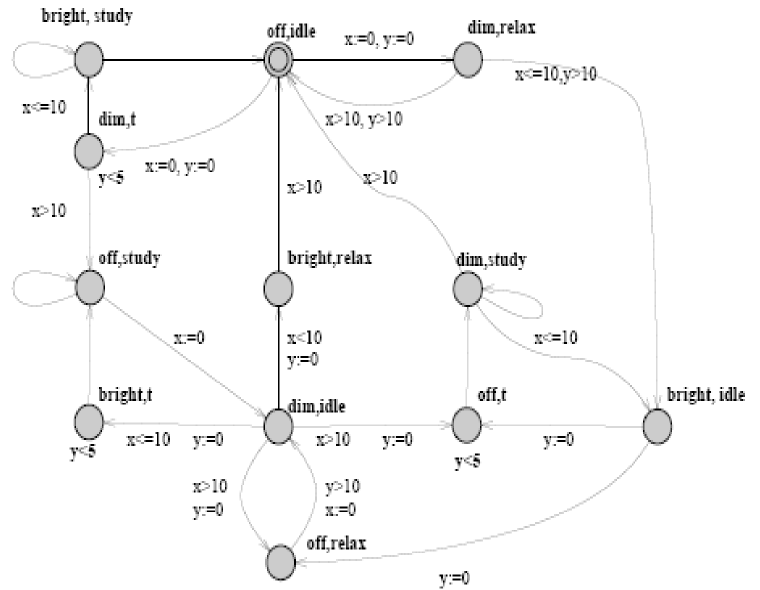


Figura 3b

La cardinalità degli stati dell'automa derivante è pari al prodotto cartesiano degli stati degli automi originali. Lo stato iniziale del secondo automa è dato dalle due locazioni iniziali dei due automi, ovvero off e idle.

Assumiamo che la u abbia il corrente valore dei clock x e y posto rispettivamente a 8 e 0.

Un esempio di delay transition è rappresentata dal cappio sul nodo dim,study:

dim,study,x=8,y=0 -> dim,study,x=9,y=1, che soddisfa la condizione sugli invarianti, che in questo caso non ci sono.

Nell'automa non sono presenti local action perché tutti gli archi sono sincronizzati mediante il canale press.

Un esempio di synchronizing action è dato dalla transizione dal nodo dim,idle al nodo bright,t: infatti

per dim,idle,x=8,y=0 -> bright,t,x=9,y=0

Abbiamo che gli archi sono sincronizzati tramite il canale press, la funzione di assegnazione u soddisfa le guardie (infatti x=8, quindi x<=10) ed anche la funzione u' soddisfa gli invarianti delle locazioni, poiché y=0 soddisfa l'invariante della locazione t che richiede y<5.

Capitolo 2
Logiche temporali e model checking per i sistemi real-time

Un contesto molto diffuso e di particolare interesse nella nostra trattazione è il problema del model checking nel quale la descrizione del sistema si basa sui suddetti timed automata, mentre le specifiche sono formule TCTL, un'estensione, corredata da clock, della logica CTL (Computation Tree Logic).

2.1 Computation Tree Logic (CTL)

La logica LTL, di cui abbiamo avuto modo di approfondire la versione proposizionale durante il corso di Metodi Formali nell'Ingegneria del Software, permette di esprimere formule riguardanti un singolo cammino di computazione in quanto basata su un modello lineare del tempo.
CTL è invece una logica modale temporale basata su un modello temporale ramificato (*branching-time*). LTL e CTL sono entrambi sottoinsiemi della più generica logica CTL*. Esistono espressioni in CTL non esprimibili in LTL e viceversa.

Definizione 2.1.1 (Sintassi di CTL) Sia *p* un atomo proposizionale elemento di un qualche insieme A di Atomi. Una formula di CTL è definita come segue:

$\phi ::=$ T | $\perp$ | p | $(\neg\phi)$ | $(\phi \wedge \phi)$ | $(\phi \vee \phi)$ | $(\phi \to \phi)$ | AXϕ | EXϕ | AFϕ | EFϕ | AGϕ | EGϕ | A$[\phi \, \mathrm{U} \, \phi]$ | E$[\phi \, \mathrm{U} \, \phi]$

Si ricorda che le formule delle logiche temporali sono interpretate sugli stati delle strutture temporali, consistenti in una tripla (S,R,L) dove S è un insieme finito di stati, $R \subset S \times S$ è una relazione di transizione ed $L : S \to (LP \to \{T, F\})$ è una funzione di etichettatura che associa ad ogni stato un'interpretazione delle lettere dell'alfabeto LP.

Definizione 2.1.2 (Semantica di CTL) Dato uno stato *q* in una struttura temporale:
$q \models$ T ,
$q \not\models \perp$,

q ⊨= p se e solo se p$\in L(q)$,
q ⊨=$\neg\varphi$ se e solo se q ⊨≠ φ ,
q ⊨= $\varphi \wedge \psi$ se e solo se q ⊨=φ e q ⊨=ψ ,
q ⊨= $\varphi \vee \psi$ se e solo se q ⊨=φ oppure q ⊨=ψ ,
q ⊨= $\varphi \rightarrow \psi$ se e solo se q ⊨=ψ qualora q ⊨=φ ,

q ⊨= AXφ se e solo se per ogni $q_1 \in \pi^q$ si ha q_1 ⊨= φ ,

q ⊨= EXφ se e solo se esiste un $q_1 \in \pi^q$ con t.c. q_1 ⊨= φ ,

q ⊨= AGφ se e solo se per ogni π^q e per ogni $q_i \in \pi^q$ si ha q_i ⊨= φ ,

q ⊨= EGφ se e solo se esiste un π^q t.c. per ogni $q_i \in \pi^q$ si ha q_i ⊨= φ ,

q ⊨= AFφ se e solo se per ogni π^q esiste un $q_i \in \pi^q$ t.c. q_i ⊨= φ ,

q ⊨= EFφ se e solo se esiste un π^q ed esiste un $q_i \in \pi^q$ t.c. q_i ⊨= φ ,

q ⊨= A$[\varphi \mathrm{U} \psi]$ se e solo se ogni π^q si ha che π^q ⊨= $\varphi \mathrm{U} \psi$, cioè se e solo se esiste un $i \geq 1$ t.c. q_i ⊨= ψ e per ogni $j = 1,...,i-1$ si ha q_j ⊨= φ ,

q ⊨= E$[\varphi \mathrm{U} \psi]$ se e solo se esiste π^q t.c. π^q ⊨= $\varphi \mathrm{U} \psi$, cioè se e solo se esiste un $i \geq 1$ t.c. q_i ⊨= ψ e per ogni $j = 1,...,i-1$ si ha q_j ⊨= φ ,

dove ricordiamo che π^q denota il suffisso del cammino π che inizia dallo stato q compreso, e imponiamo che q_1 rappresenti il secondo stato nel cammino.

La logica CTL ricorre in genere all'utilizzo degli operatori visti nella trattazione della logica LTL (Next, Future, Globally, Until), ma introduce anche gli operatori A (All paths) ed E (Exists a path) che quantificano sui possibili cammini, rispettivamente "Per tutti i cammini" ed "Esiste (almeno) un cammino". Questa logica nasce dall'osservazione dell'impossibilità per LTL di quantificare sulle tracce.
Si consideri la struttura temporale in figura 1.

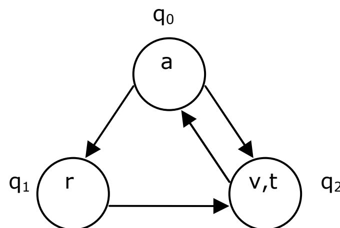


Figura 1

Essa può modellare in maniera molto semplificata l'avvicinarsi ad un incrocio con semaforo, nel quale i simboli r e v rappresentano rispettivamente le circostanze in cui il semaforo sia rosso o verde, mentre il simbolo a l'avvicinamento all'incrocio e il simbolo t il transito attraverso l'incrocio. La frase in linguaggio naturale "Esiste la possibilità che il semaforo non sia rosso" non è esprimibile in LTL. In CTL invece esso si scriverà:
- EF$\neg r$

L'utilità pratica di CTL sta nel poter esprimere tramite formule proprietà di algoritmi e protocolli la cui verifica è fondamentale, come le proprietà di liveness e safety.

2.2 Timed Computation Tree Logic (TCTL)

La differenza principale tra TCTL e CTL sta nel contemplare i clock nella definizione di TCTL e nel model checking.

Definizione 2.2.1 (Sintassi di TCTL) Sia *p* un atomo proposizionale elemento di un qualche insieme A di Atomi e $Cl = \{x, y, ...\}$ un insieme di clock. Una formula di TCTL è definita come segue:

$\phi ::=$ T | $\perp$ | p | $(\neg\phi)$ | $(\phi \wedge \phi)$ | $(\phi \vee \phi)$ | $(\phi \to \phi)$ | AXϕ | EXϕ | AFϕ | EFϕ | AGϕ | EGϕ | A$[\phi$ U $\phi]$ | E$[\phi$ U $\phi]$ | $x\ \underline{in}\ \phi$ | $x \sim k$

dove ~ può essere un qualsiasi operatore di confronto tra $\{=, \leq, <, \geq, >\}$, e $k \in \mathrm{N}$.

Definizione 2.2.2 (Semantica di TCTL) Dato uno stato *q* in una struttura temporale e una funzione di valutazione $v : Cl \to \mathrm{N}$ che associa ad ogni variabile di clock in φ un numero naturale:

$q, v \models$ T ,
$q, v \not\models \perp$,
$q, v \models$ p se e solo se p $\in L(q)$,
$q, v \models \neg\varphi$ se e solo se $q, v \not\models \varphi$,
$q, v \models \varphi \wedge \psi$ se e solo se $q, v \models \varphi$ e $q, v \models \psi$,
$q, v \models \varphi \vee \psi$ se e solo se $q, v \models \varphi$ oppure $q, v \models \psi$,
$q, v \models \varphi \to \psi$ se e solo se $q, v \models \psi$ qualora $q, v \models \varphi$,
$q, v \models$ AXφ se e solo se per ogni $q_1 \in \pi^q$ si ha $q_1, v + d \models \varphi$,
$q, v \models$ EXφ se e solo se esiste un $q_1 \in \pi^q$ con t.c. $q_1, v + d \models \varphi$,
$q, v \models$ AGφ se e solo se per ogni π^q e per ogni $q_i \in \pi^q$ si ha $q_i, v \models \varphi$,
$q, v \models$ EGφ se e solo se esiste un π^q t.c. per ogni $q_i \in \pi^q$ si ha $q_i, v \models \varphi$,
$q, v \models$ AFφ se e solo se per ogni π^q esiste un $q_i \in \pi^q$ t.c. $q_i, v \models \varphi$,
$q, v \models$ EFφ se e solo se esiste un π^q ed esiste un $q_i \in \pi^q$ t.c. $q_i, v \models \varphi$,
$q, v \models$ A$[\varphi$ U $\psi]$ se e solo se ogni π^q si ha che $\pi^q, v \models \varphi$ U ψ , cioè se e solo se esiste un $i \geq 1$ t.c. $q_i, v \models \psi$ e per ogni $j = 1, ..., i-1$ si ha $q_j, v \models \varphi$,
$q, v \models$ E$[\varphi$ U $\psi]$ se e solo se esiste π^q t.c. $\pi^q, v \models \varphi$ U ψ , cioè se e solo se esiste un $i \geq 1$ t.c. $q_i, v \models \psi$ e per ogni $j = 1, ..., i-1$ si ha $q_j, v \models \varphi$,
$q, v \models x\ \underline{in}\ \phi$ se e solo se $q, v[x \leftarrow 0] \models \varphi$,
$q, v \models x \sim k$ se e solo se $v(x) \sim k$,

dove $d \in Time(\pi_{|i})$, che denota il tempo impiegato per raggiungere q_i a partire da q_0 lungo il cammino π , e $v[x \leftarrow 0]$ è la valutazione v in cui $x = 0$.

La novità introdotta dalla logica TCTL è la possibilità di esprimere vincoli temporali sui clock presenti nella formula. Ad esempio le formule:

(1) $z\ \underline{in}$ A$[\varphi \wedge z \leq 7$ U $\psi]$

(2) $z\ \underline{in}$ EF$(z < 5 \wedge \phi)$

significano rispettivamente:

(1) lungo ogni possibile cammino, φ rimane vera fino a quando entro 7 unità di tempo ψ diventa valida;

(2) la proprietà ϕ diverrà valida entro 5 unità di tempo.

2.3 Model Checking

Il model checking è una preziosa tecnica per la verifica di proprietà, espresse in opportuna logica, di un sistema con un insieme finito di stati.

È stato provato che il model checking applicato ai sistemi real-time è decidibile per un certo numero di logiche temporali, tra cui TCTL. In particolare per questa logica esso risulta PSPACE-completo, in quanto è stato dimostrato essere in PSPACE ed essere PSPACE-hard anche per una fissata struttura temporale.

L'osservazione cruciale fatta da Alur, Courcoubetis e Dill nonché fondamento della decidibilità del model checking, è, sommariamente, l'effettivo partizionamento di un insieme (eventualmente infinito) di assegnazioni di valori di clock in un numero finito di *regioni* in maniera tale che le assegnazioni all'interno di una regione generino stati che soddisfino le stesse proprietà logiche. Ogni regione essenzialmente consiste nell'insieme di stati equivalenti, nel senso che possono evolvere nelle stesse regioni nel futuro.

L'applicazione pratica del model checking in tale scenario, tuttavia, comporta il problema della potenziale esplosione combinatoria degli stati, risolvibile ricorrendo a strutture cosiddette *simboliche*, come i Binary Decision Diagram o le Difference Bound Matrix. Mentre la natura indipendente dalle proprietà delle regioni comporta un partizionamento fine e cospicuo dell'insieme delle assegnazioni di clock, una tecnica simbolica integra nelle partizioni la particolare proprietà che deve essere verificata, evitando lo sviluppo di stati che non verrebbero comunque presi in considerazione.

Per una trattazione più dettagliata del concetto di *regioni*, si rimanda ai riferimenti **[19]** e **[3]**.

2.4 Reachability Analysis

La nuova tecnica è sviluppata per una logica meno espressiva di TCTL, ma sufficiente per scopi pratici, guadagnando in efficienza nel model checking. In questo modo, il problema si riduce alla verifica delle proprietà di reachability (*Reachability Analysis*), ed in particolare la raggiungibilità di combinazioni date di nodi e vincoli sui clock a partire da una configurazione iniziale. Tali proprietà sono della forma:

$$\phi ::= \text{AG}\,\beta \mid \text{EF}\,\beta \qquad \beta ::= a \mid \beta_1 \wedge \beta_2 \mid \neg\beta$$

dove a è una formula atomica che può essere un vincolo di clock o una locazione.

Intuitivamente, affinché AG β sia soddisfatta, tutti gli stati raggiungibili devono soddisfare β. Analogamente, affinché EF β sia soddisfatta, alcuni stati raggiungibili devono soddisfare β. Formalmente, la relazione di soddisfacibilità tra gli stati e le formule sono definiti come segue:

$$\langle \bar{l}, v \rangle \models \text{EF}\,\beta \Leftrightarrow \exists \langle \bar{l}', v' \rangle \text{ t.c. } \langle \bar{l}, v \rangle \xrightarrow{*} \langle \bar{l}', v' \rangle \wedge \langle \bar{l}', v' \rangle \models \beta$$

$$\langle \bar{l}, v \rangle \models \text{AG}\, \beta \Leftrightarrow \forall \langle \bar{l}', v' \rangle \text{ t.c. } \langle \bar{l}, v \rangle \xrightarrow{*} \langle \bar{l}', v' \rangle \wedge \langle \bar{l}', v' \rangle \models \beta$$

Un'importante congettura (vedi riferimento **[7]**) è la traducibilità in problemi di raggiungibilità anche delle proprietà cosiddette di *bounded liveness*, proprietà che verificano il raggiungimento di uno scopo entro un massimo delay.
La raggiungibilità è un problema PSPACE-completo.

Capitolo 3

Panoramica su Uppaal e timed automata in Uppaal

3.1 Uppaal

Uppal 4.0 è un tool per modellare, simulare e verificare in modo automatico proprietà dei sistemi real-time modellati come una rete di timed automata.

E' stato sviluppato dalla collaborazione tra il centro BRICS, Basic Research in Computer Science, dell'Università di **Aal**borg ed il Department of Computing Systems dell'Università di **Upp**sala nella primavera 1995.

Uppaal è utile per sviluppare quei sistemi che possono essere modellati come una collezione di processi non deterministici che comunicano attraverso canali e/o variabili condivise. Le aree di applicazione tipiche sono quindi i sistemi real-time ed i protocolli di comunicazione nei quali il fattore tempo è determinante.

Le caratteristiche principali alle quali hanno puntato gli sviluppatori di Uppaal, e che sembrano essere state raggiunte pienamente, sono state l'efficienza, la facilità di utilizzo ed una sintassi e semantica abbastanza intuitiva.

3.2 Architettura di Uppal

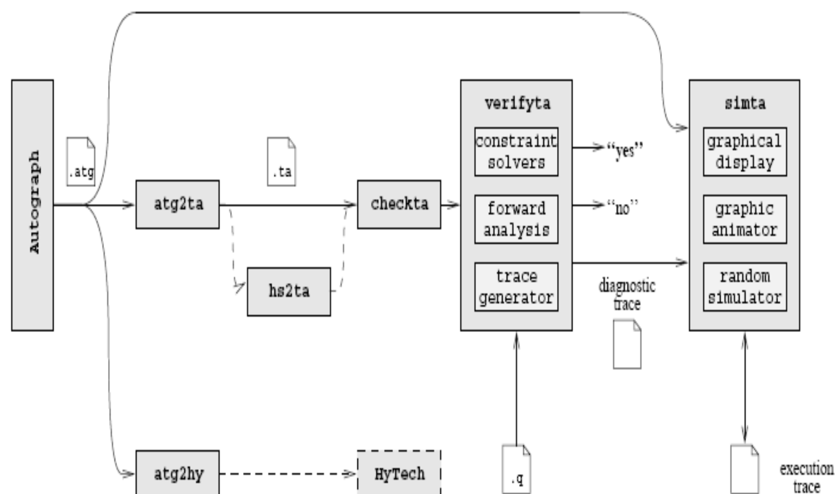


Figura 1: Architettura Uppaal

Le parti principali che costituiscono e compongono il tool Uppaal sono tre:

1. linguaggio di modellazione
2. simulatore
3. model checker

3.2.1 Linguaggio di modellazione

Il linguaggio di modellazione di Uppaal consente, come già detto, di descrivere i sistemi real-time come reti di timed automata scegliendo di poter utilizzare due modalità, una grafica ed una testuale.

La sintassi delle espressioni di Uppaal coincide con la sintassi di C/C++.

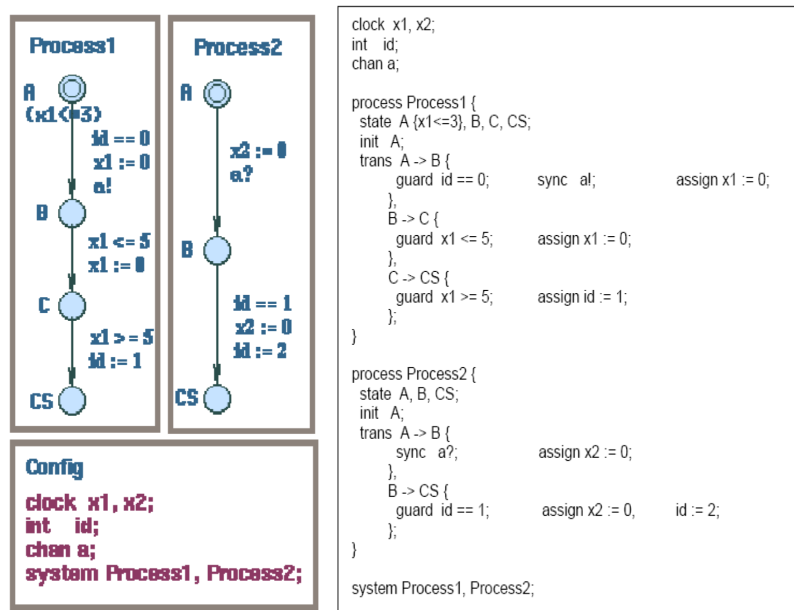


Figura 2: (a) Modello Grafico (b) Modello Testuale

<u>Modalità grafica:</u>

Autograph, la cui interfaccia è mostrata in figura 3, è un editore grafico che consente di disegnare reti di automi producendo un file in formato .atg.

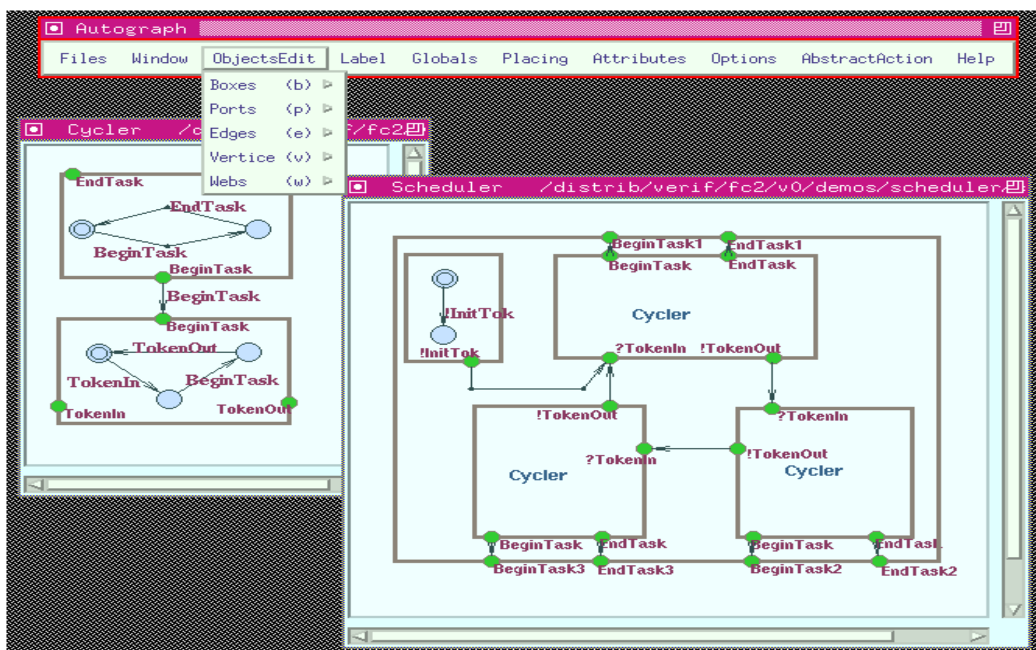


Figura 3: Autograph

Per poter disegnare reti di automi occorrono seguire alcune regole sintattiche, figura 2(a), come ad esempio che differenti automi appartenenti alla stessa rete siano disegnati all'interno di box etichettate con il loro nome e che venga fornita una descrizione testuale del sistema dove si indica che quei processi, automi, appartengono a quel dato sistema. Inoltre ogni stato di ogni automa può contenere una lista di invarianti ed ogni transizione tra due stati di un automa può essere etichettata con una guardia, una sincronizzazione o un'assegnazione di variabile. E' possibile inoltre dichiarare variabili di clock globali, variabili di dati e canali di sincronizzazione.
Come mostra la figura 1, il file ottenuto utilizzando Autograph viene dato in pasto ad un una compilatore automatico che effettua la trasformazione dal formato grafico al formato testuale.

Modalità testuale:

La figura 2(b) mostra lo stesso sistema nel formato testuale. La corrispondenza tra le due modalità è evidente: ogni automa in figura 2(a) diventa un processo in figura 2(b) contenente lo stato iniziale, una lista di dichiarazioni di stati con invarianti ed una lista di transizioni con guardie, sincronizzazioni ed assegnazioni.
La lista di dichiarazioni di stati dichiara tutti gli stati dell'automa ed ogni stato può essere seguito da una lista di invarianti racchiusi tra parentesi tonde:

state stateName(invariant, invariant…), stateName(invariant, invariant,…)…

Lo stato iniziale è dichiarato con:

init stateName

Ogni transizione infine è dichiarata specificando lo stato iniziale e quello finale:

trans stateName→stateName

ed eventualmente una lista di guardie, sincronizzazioni ed assegnazioni:

{ guard…
synchronization…
assignments… }

All'inizio del file inoltre sono dichiarate le variabili di clock, le variabili dei dati, i canali di sincronizzazione e la dichiarazione dei processi che compongono il sistema.

La figura 4 mostra la sintassi della modalità testuale

```
<global variable declarations>

process <automaton name>
{
        state <state declaration list>
        init <initial state>
        trans [ <start state> ] -> <end state>
        {
                [<guard list>]
                [<synchronization list>]
                [<assignment list>]
        },
        [ <start state> ] -> <end state>
        {
                [<guard list>]
                [<synchronization list>]
                [<assignment list>]
        },
        ...
}

<other automaton definition >
...

<automaton declarations>
```

Figura 4: Sintassi

La figura 1 mostra come il convertitore atg2ta automaticamente crei a partire da un file .atg il rispettivo file testuale in formato *.ta* semplicemente perché questo disegno si riferisce a versioni precedenti, Uppaal 3.0, a quella da noi utilizzata. Uppaal 4.0 infatti non solo supporta file .ta ma anche il formato XML e XTA sempre per la modellazione del sistema, un formato dedicato per i file di query ed il formato XTR, già presente in Uppaal 3.0, per memorizzare le tracce relative al modello realizzato.

Formato XTA, già presente nella versione 3.0, è utilizzato per memorizzare sistemi sintatticamente corretti ed informazioni grafiche sul sistema ed in Uppaal 4.0 è stato esteso per poter supportare le selezioni sugli archi. File scritti in questo formato presentano l'estensione .xta.

Formato TA è un sottoinsieme dell'XTA ma a differenza di questo non mantiene nessuna informazione grafica sul sistema e non supporta il concetto di template. Uppaal 4.0 non produce file di questo tipo ma è in grado di leggerli.

Formato XML

E' il formato utilizzato da Uppaal 4.0 ed i file che utilizzano tale formato hanno l'estensione .xml. Template, locazioni, archi ed etichette sono descritti usando *tag* così come ogni caratteristica riguardante informazioni grafiche come colori e note sui nodi ed archi e l'attributo di selezione sono specificati usando l'attributo colore e due nuovi tipi di etichette. Questo file è anche supportato dal motore di verifica ed ogni nuovo progetto creato è salvato per default in questo formato.

Qua sotto mostriamo il file .xml del progetto che sarà al centro del nostro studio ovvero DHCP che verrà nell’ultimo capitolo analizzato.

```
<?xml version="1.0" encoding="utf-8"?><!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat
System 1.1//EN' 'http://www.it.uu.se/research/group/darts/uppaal/flat-
1_1.dtd'><nta><declaration>//dichiarazione variabili globali

const int NServer = 3;

typedef int[0,NServer-1]id_s;

//dichiarazione canali broadcast
broadcast chan discover;
broadcast chan requestall;

//dichiarazione canali
chan request[NServer];
chan ack[NServer];
chan nack[NServer];
chan offer[NServer];
const int leasetime = 3600;

</declaration><template><name x="5" y="5">Client</name><declaration>//DICHIARAZIONI
LOCALI AL CLIENT
clock x;

const int Tsel =10;
const int Treq =15;
const int T1=1800; // leasetime *0.5;
const int T2=3150; //leasetime * 0.875;
id_s my_s;

int[0,1] rec = 0;

void receiveoffer(id_s s) {
       if(rec==0) {
              my_s = s;
              rec=1;}}

void update(id_s s) {
       my_s = s;
}

... ... ...

</template><system>// Place template instantiations here.
//Process = Template();

// List one or more processes to be composed into a system.

system Client,Server;</system></nta>
```

Per completezza di trattazione definiamo anche gli altri due formati ovvero *query file* e *file XTR*.

Query file: I file in questo formato hanno l’estensione .q e la loro struttura prevede semplicemente un elenco di tutte le query con i commenti relativi.

Formato XTR: i file in questo formato hanno l’estensione .xtr e sono legati al modello dal quale sono generati.

Libutap, il parser utilizzato da Uppaal, mediante la libreria tracer legge i file .xtr e li traduce in un formato leggibile.

3.2.2 Simulatore

Simta è il modulo Uppaal che consente di simulare il sistema.

La simulazione permette all'utente di simulare in modo interattivo e grafico il comportamento dinamico del sistema. A differenza del model-checker, che come vedremo esplora tutto lo spazio degli stati del sistema raggiungibili esaminando tutti i comportamenti di questo, il simulatore esplora solamente una particolare *traccia* di esecuzione, *execution trace,* ovvero una sequenza di stati del sistema. Questo rappresenta una forma rudimentale e poco costosa di fault detection.

E' possibile utilizzare questo modulo fornendo in input anche la diagnostic trace fornita dal verifyta per avere una rappresentazione grafica e interattiva della sequenza che porta il sistema in errore.

Componenti fondamentali sono:

control part: è usata per scegliere e selezionare le transizioni abilitate, selezionare una traccia salvata o crearne una nuova ed infine utilizzare la simulazione casuale;

variable view: mostra i valori delle variabili intere ed i vincoli di clock.

E' importante sapere che ogni transizione in Uppaal non corrisponde ad un colpo di clock. I clock infatti in Uppaal vengono rappresentati come un intervallo nel quale il clock si trova ad un certo periodo per un certo processo. Ad esempio, Server(2).x in [0,25], significa che il clock x del processo Server(2) si trova tra 0 e 25.

system view: mostra tutti gli automi istanziati e le locazioni attive dello stato corrente.

message sequence chart: mostra la sincronizzazione tra i differenti processi e le locazioni attive ad ogni passo per ogni processo.

Il simulatore può essere usato in tre modi:

- modalità manuale: l'utente può eseguire il sistema manualmente e scegliere quali transizioni fare
- modalità random: è il sistema a scegliere quali transizioni eseguire e quando.
- selezione: l'utente può eseguire una traccia, salvata o caricata dal verificatore, per vedere come certi stati sono raggiungibili.

3.2.3. Verificatore

Il modulo verifyta è utilizzato da Uppaal per effettuare model checking.

Prende in ingresso una rete di automi espressa nel formato testuale, una formula e restituisce:

- SI se la formula è soddisfatta,
- NO se la formula non è soddisfatta.

Durante la fase di verifica di una proprietà può anche essere prodotto una *diagnostic trace* che illustra perché una particolare proprietà sia stata soddisfatta oppure no. Questa informazione può essere certamente considerata un'informazione diagnostica nel senso che può risultare utile, in caso di errore del sistema, per la fase di debug.

Il verificatore fornisce anche delle opzioni per scegliere l'algoritmo di ricerca, ottimizzazioni e strutture dati.

3.3 Timed Automata in Uppaal

Questa sezione ha il compito di illustrare come le reti di timed automata vengano realizzate in Uppaal e le caratteristiche aggiuntive introdotte.

3.4 Sintassi

In Uppaal un modello di sistema è una rete di processi descritti come timed automata estesi con variabili intere oltre ai vincoli di clock messi in parallelo e definiti nel *system declaration*. Ogni processo è istanziato con un template parametrizzato cui è possibile associare le proprie variabili locali, clock e funzioni. Il sistema fornisce inoltre canali o variabili condivise per la comunicazione tra processi.

Il modello del sistema è quindi diviso in tre parti fondamentali:

- dichiarazione di variabili locali e globali,
- template dei processi,
- definizione del sistema.

3.4.1 Dichiarazioni di variabili locali e globali

Le dichiarazioni di variabili possono essere locali al processo o globali e contenere dichiarazioni di clock, canali, interi limitati o non, array e tipi.

Esistono quattro tipi predefiniti:

int che per default va da [-32768,32767]

bool che può avere il valore falso o vero equivalente al valore intero 0 o 1 come in C

chan che possono essere dichiarati anche urgenti o broadcast

clock per la dichiarazione di variabili di clock.

Gli interi limitati (*bounded integer*) sono dichiarati come *int[min, max] name* dove min e max sono il limite inferiore e superiore rispettivamente dell'intervallo. Tali limiti sono controllati durante la verifica e la violazione di un limite a run-time porta in uno stato non valido.

Le costanti sono dichiarate scrivendo *const name value* e per definizione di costante non possono essere modificate e devono avere valore intero.

E' possibile inoltre creare array di variabili di clock, di costanti, di canali e di interi definiti aggiungendo una dimensione al nome di variabile. Ad esempio chan c[4] è un array di canali di dimensione 4.

3.4.1.1 Esempio di dichiarazione di variabile:

- const int a=1; costante a con valore intero 1
- bool b[8]; array b di booleani contenente 8 elementi
- int[0,100] c=5; una variabile intera c che va da 0 a 100 inizializzata a 5
- chan d; canale d
- urgent chan e; canale e urgente
- broadcast chan f; canale f broadcast
- clock x,y; due variabili di clock x ed y

3.4.2 Template dei processi

Un template di un processo è costituito da nodi ed archi, può contenere parametri e dichiarazioni locali.

Ogni locazione di un timed automata è rappresentata con un cerchio (la locazione iniziale in particolare due cerchi) che può avere un nome associato, connesso agli adiacenti tramite archi.

Esistono due tipi di locazioni:

Urgent location: queste locazioni sono indipendenti dall'avanzare dei clock dichiarati, ovvero quando il processo si trova in questa locazione il tempo non scorre. Semanticamente la locazione urgente

corrisponde ad aggiungere un clock x extra che è resettato su ogni arco entrante e un invariante x<=0 sulla locazione.
Committed location: come la locazione urgente "blocca" il tempo, obbligando inoltre la transizione su un arco verso un'altra locazione adiacente. In altre parole, il sistema dovrà uscire dallo stato committed prima che ricominci ad avanzare il tempo.

Ogni nodo inoltre può essere etichettato con un invariante: una congiunzione di semplici condizioni sui clock, differenze tra clock ed espressioni booleane che non coinvolgono clock. Se l'invariante è vero il sistema può decidere di rimanere nello stato o muoversi; nel momento in cui diventa falso deve necessariamente uscire da quello stato.
Esempi:
x<=2; il clock x deve essere minore od uguale a 2 e quindi lo stato con quell'invariante dovrà effettuare la transizione verso un altro stato prima che diventi maggiore di 2.
x<y; il clock x deve essere minore del clock y.

Gli archi di un timed automata sono rappresentati con delle "linee" tra i nodi ed è possibile specificarne quattro campi opzionali:
select: assegnano non deterministicamente un identificatore ad un valore di un certo tipo che appartiene ad un certo range e tale identificatore può essere utilizzato come variabile dalle altre etichette dell'arco.
Il tipo concesso è un bounded integer.
Esempio:
Select: i: int[0,3] //ad i viene assegnato un intero compreso tra 0 e 3
Synchronization: a[i]? // identificatore i è utilizzato come indice di array per sapere quale canale deve essere sincronizzato

Guardia: una guardia è l'unione di vincoli sui clock e sui dati dove i vincoli sui clock sono della forma $x \sim n$, $x - y \sim n$ con n numero naturale e $\sim \in \{\leq, \geq, =, >, <\}$ e i vincoli sui dati della forma $i \sim j$ o $i - j \sim k$ con k intero. Possono essere semplici condizioni sui clock, differenze tra valori di clock o condizioni booleane su variabili intere che non coinvolgono clock. Un arco è abilitato per uno stato se e solo se la guardia sull'arco è vera.

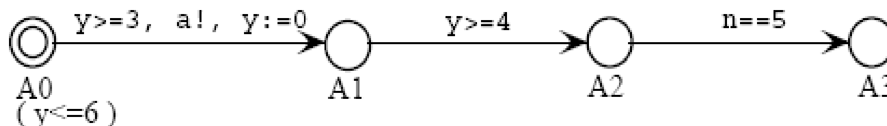


Figura 5: esempio di guardia

La figura 5 mostra una guardia, ovvero un vincolo di clock y>=3 sull'arco A0→A1. Ciò significa che l'arco potrà essere preso solo quando il valore del clock y sarà maggiore o uguale a 3, mentre l'arco tra A2 ed A3 può essere preso solamente quando il valore della variabile intera n è uguale a 5.
Nel caso in cui ci sia più di una guardia per arco allora viene valutata l'unione di tutte le guardie.

Sincronizzazione: i processi mediante i canali ed archi etichettati con azioni complementari possono sincronizzarsi su un canale comune.
Le azioni complementari sono della forma **e?** oppure **e!**, dove e è il canale, e? indica il ricevente ed e! il mittente, ed è un'espressione che non comporta side-effect. Quando due processi si sincronizzano entrambi gli archi etichettati con le azioni complementari vengono presi allo stesso tempo e se nell'arco dell'azione di sender è presente un update questo viene eseguito prima di un eventuale update sull'arco etichettato con e?.
Come già detto, esistono due tipologie opzionali di canali:
- Urgent channel: sono canali dichiarati utilizzando il prefisso urgent chan name e fanno in modo che, se la transizione sul canale urgente è abilitato, non ci siano ritardi ovvero che la transizione di sincronizzazione venga effettuata subito. Archi che consentono la sincronizzazione urgente non possono avere specificati vincoli sul tempo come le guardie.
- Broadcast channel: sono canali dichiarati utilizzando il prefisso broadcast chan name e consentono una sincronizzazione tra processi del tipo uno-a-molti ovvero un sender può sincronizzarsi con un numero arbitrario di receiver.
Si ricorda che Uppaal consente anche di effettuare una sincronizzazione mediante l'utilizzo di variabili intere condivise.

Update: consente di assegnare valori a variabili di clock e a variabili di dati.
L'operatore = può essere utilizzato per assegnare un valore ad una variabile di clock, un intero o un booleano, mentre gli altri operatori come +=,-= ecc sono riservati solo a variabili booleane ed intere.
E' da notare inoltre che alla destra dell'operatore possono trovarsi solamente costanti o variabili stesse.
Esempi:
x=0; il clock x viene azzerato.
x=1,y=2*x; la variabile intera x è settata ad 1 mentre y a 2 infatti le assegnazioni sono valutate in maniera sequenziale.

Nella seguente sezione si definirà il modello, ovvero quali sono i processi concorrenti che ne fanno parte, le variabili locali e globali e i canali.

3.5 Semantica

Gli stati di un modello Uppaal sono della forma $\{(L,v) \mid v$ soddisfa $I(L)\}$, dove L è un vettore di controllo che indica per ogni componente della rete di timed automata quale sia il corrente nodo di controllo e v è un'assegnazione che mappa il valore corrente per ogni clock del sistema e le variabili intere ai loro valori e I una funzione che mappa le locazioni ed il vettore di controllo delle locazioni agli invarianti.
Lo stato iniziale è lo stato nel quale tutti i processi del modello sono nella locazione iniziale, tutte le variabili hanno il loro valore iniziale e tutti i clock sono a zero.

Le transizioni tra stati in un modello Uppaal possono essere di due tipi:
Dalay transitions: modella il passaggio del tempo senza cambiare le locazioni correnti.
Abbiamo una delay transition $\langle L,v\rangle \xrightarrow{d} \langle L,v'\rangle$, dove d è un reale non negativo, se e solo se:

v' = v + d, dove *v+d* è ottenuto incrementado di *d* tutti i clock.

Per ogni $0 \leq d' \leq d$: *v + d'* soddisfa *I*(*L*)

Tale transizione si può verificare solo se *L* non contiene né locazioni committed né urgenti.

Per tutte le locazioni *l* in *L* and per tutte le locazioni *l'* (non necessariamente in *L*), se c'è un arco da *l* ad *l'* allora:

- questo arco non si sincronizza su un canale urgente, oppure
- questo arco si sincronizza su un canale urgente ma per tutti $0 \leq d' \leq d$ abbiamo che *v + d'* non soddisfa la guardia sull'arco.

Fino a quando nessuno degli invarianti sui nodi di controllo è violato nello stato corrente il tempo deve scorrere senza produrre effetti sul vettore dei nodi di controllo ed incrementando tutti i valori dei clock di una stesso incremento.

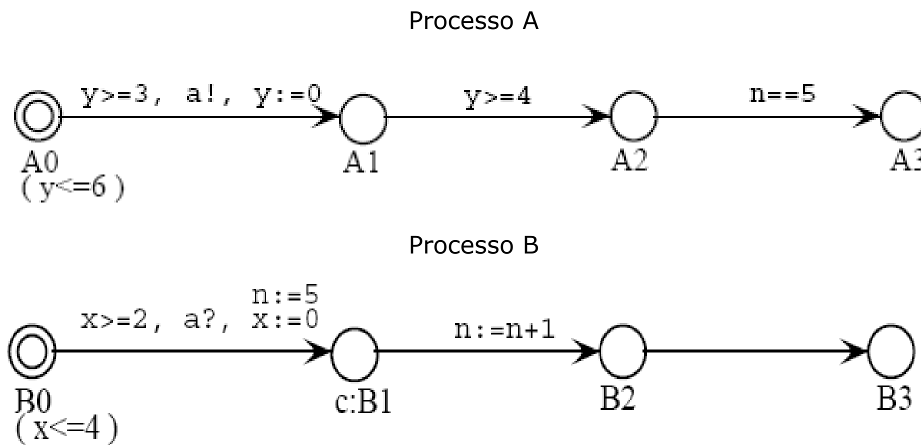


Figura 6: rete di timed automata

La figura 6 mostra che partendo dallo stato iniziale ((A0,B0),x=0,y=0,n=0) il tempo può scorrere per 3.5 unità di tempo raggiungendo lo stato ((A0,B0),x=3.5,y=3.5,n=0) però tuttavia non può scorrere per 5 unità poiché violerebbe l'invariante x<=4 in B0.

<u>Action Transition:</u> per questo tipo di transizioni è fondamentale la sincronizzazione sugli archi.

Se due archi di due componenti differenti etichettati in maniera complementare sono attivi in un certo stato allora possono sincronizzarsi.

Nella figura 6 ad esempio supponiamo di stare nello stato ((A0,B0), x=3.5,y=3.5,n=0) i due processi possono sincronizzarsi sul canale **a** andando nel nuovo stato ((A1,B1),x=0,y=0,n=5).

Da notare inoltre che se un componente ha un arco interno abilitato allora tale arco può essere preso senza alcuna sincronizzazione. Per esempio se il sistema si trova nello stato ((A1,B1),x=0,y=0,n=5) il processo B può passare nello stato ((A1,B2),x=0,y=0,n=6) senza sincronizzarsi con **a**.

È importante osservare che in presenza di locazioni committed in L, cioè se uno degli automi della rete si trova in uno stato committed, la transizione deve condurre l'automa in uno stato adiacente prima che avvenga qualsiasi altra transizione (anche di tipo delay).

Esistono tre tipi di action transition:

<u>Internal Transitions</u>

Abbiamo una transizione $\langle L, v \rangle \xrightarrow{*} \langle L', v' \rangle$ se c'è un arco *e*=(*l*,*l'*) tale che:

- Non ci sono etichette di sincronizzazione su *e*;

- V soddisfa la guardia su e;
- $L' = L[l'/l]$;
- v' è ottenuto da v eseguendo l'update su e;
- v' soddisfa $I(L')$;
- né l è committed, né un altra locazione in L lo è.

Binary Synchronisations

Abbiamo una transizione $\langle L, v \rangle \xrightarrow{*} \langle L', v' \rangle$ se ci sono due archi $e_1=(l_1,l_1')$ and $e_2=(l_2,l_2')$ in due processi differenti tali che:

- $e1$ ha un'etichetta di sincronizzazione $c!$ and e_2 ha una etichetta di sincronizzazione $c?$, dove c è un canale binario;
- v soddisfa le guardie di e_1 e e_2.
- $L' = L[l_1'/l_1, l_2'/l_2]$
- v' è ottenuto da v prima eseguendo l'update su e_1 e poi su e_2.
- v' soddisfa $I(L')$
- né l_1 o l_2 o entrambe le locazioni sono committed né nessuna altra locazione in L è committed.

Broadcast Synchronisations

Assumiamo un ordine di processi p_1, p_2, ... p_n dato dall'ordine dei processi nel system declaration statement. Abbiamo una transizione $\langle L, v \rangle \xrightarrow{*} \langle L', v' \rangle$ se c'è un arco $e=(l,l')$ e m archi $e_i=(l_i,l_i')$ con $1 \le i \le m$ tale che:

- Gli archi e, e_1, e_2, ..., e_m sono in processi differenti
- e_1, e_2, ... e_m sono ordinati in accordo con l'ordine dei processi p_1, p_2,... p_n.
- e ha una etichetta di sincronizzazione $c!$ e e_1, e_2, ... e_m hanno le etichette di sincronizzazione $c?$, dove c è un canale broadcast.
- v soddisfa le guardie su e, e_1, e_2, ... e_m.
- Per tutte le locazioni l in L nessuna è sorgente di uno degli archi e, e_1, e_2, ... e_m, tutti gli archi da l o non hanno una etichetta di sincronizzazione $c?$ o v non soddisfa la guardia sull'arco.
- $L' = L[l'/l, l_1'/l_1, l_2'/l_2, \ldots l_m'/l_m]$
- v' è ottenuto da v prima eseguendo l'update su e poi gli update su ei seguendo l'ordine di i.
- v' soddisfa $I(L')$
- né una o più di una delle locazioni l, l_1, l_2, ... l_m sono committed o nessun'altra locazione in L è committed

Importante:

- In uno stato dove due componenti devono sincronizzarsi su un canale urgente non è concesso nessun tipo di delay. Se nel sistema descritto in figura 7 il canale a è urgente, il tempo non può scorrere di 3.5 unità di tempo dallo stato iniziale ((A0,B0),x=0,y=0,n=0) e la sincronizzazione su a è possibile

nello stato ((A0,B0),x=3,y=3,n=0).

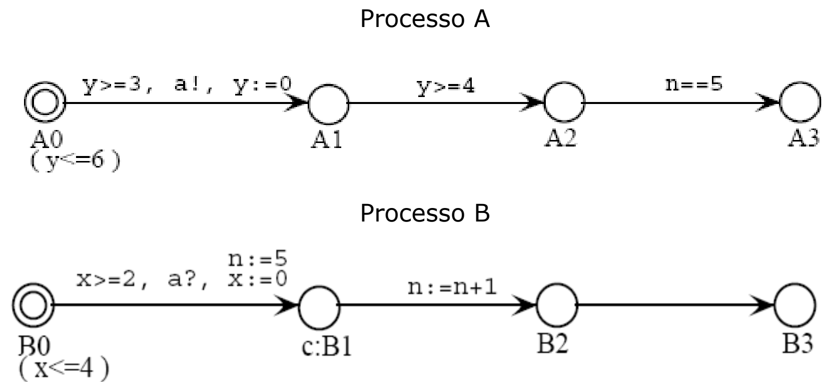


Figura 7

- Se in uno stato uno dei componenti è in un nodo di controllo committed, allora non è concesso che avvenga nessun delay e tutte le action transition devono riguardare quel componente.
  Nell'esempio in figura 7 se nello stato ((A1,B1),x=0,y=0,n=5) B1 è committed, allora senza nessun ritardo le prossima transizione deve riguardare il componente B. Lo stato successivo della rete potrà essere ((A1,B1),x=0,y=0,n=6) , garantendo che le prime due transizioni di B vengano eseguite atomicamente.

3.6 Model Checking in UPPAAL

I modelli descritti in Uppaal possono essere verificati specificando le proprietà desiderate tramite l'*Uppaal verification language*. Esso è sintatticamente indipendente dal linguaggio di modellazione ed è letto separatamente dal model checker.

Nella verifica dei modelli proprietà come correttezza degli invarianti, assenza di deadlock e livelock sono controllate automaticamente. Tuttavia l'utente può specificare proprietà addizionali del tipo

$$E<>\beta \mid A[]\beta \mid E[]\beta \mid A<>\beta,\ \beta ::= \text{atomic} \mid \beta_1 \sim \beta_2 \mid \text{not } \beta \mid (\beta)$$

dove atomic è una formula atomica che può essere un vincolo di clock o sui dati, o una proposizione riguardante un particolare automa in un particolare stato e ~ può essere or, and, implica, $>, \geq, <, \leq$ o ==.

[] corrisponde al simbolo della logica G, ovvero l'operatore *Globally*, mentre <> corrisponde a F, ovvero *Future*.

Le proprietà supportate saranno limitate a cinque tipologie:

- E<> p è vera se e solo se c'è esiste un cammino che conduce ad uno stato che soddisfa *p*.
- A[] p è vera se e solo se ogni stato raggiungibile soddisfa p (*reachability*).
- E[] p è vera se e solo se esiste un cammino per il quali *p* è costantemente verificata in tutti gli stati indefinitamente.
- A<> p è vera se e solo se tutti i possibili cammini prima o poi raggiungono uno stato che soddisfa p (*liveness*).
- p --> q denota una proprietà "conduce a" (*leads to*) che significa che ogni qualvolta p si verifica costantemente, prima o poi q diverrà anche essa costante.

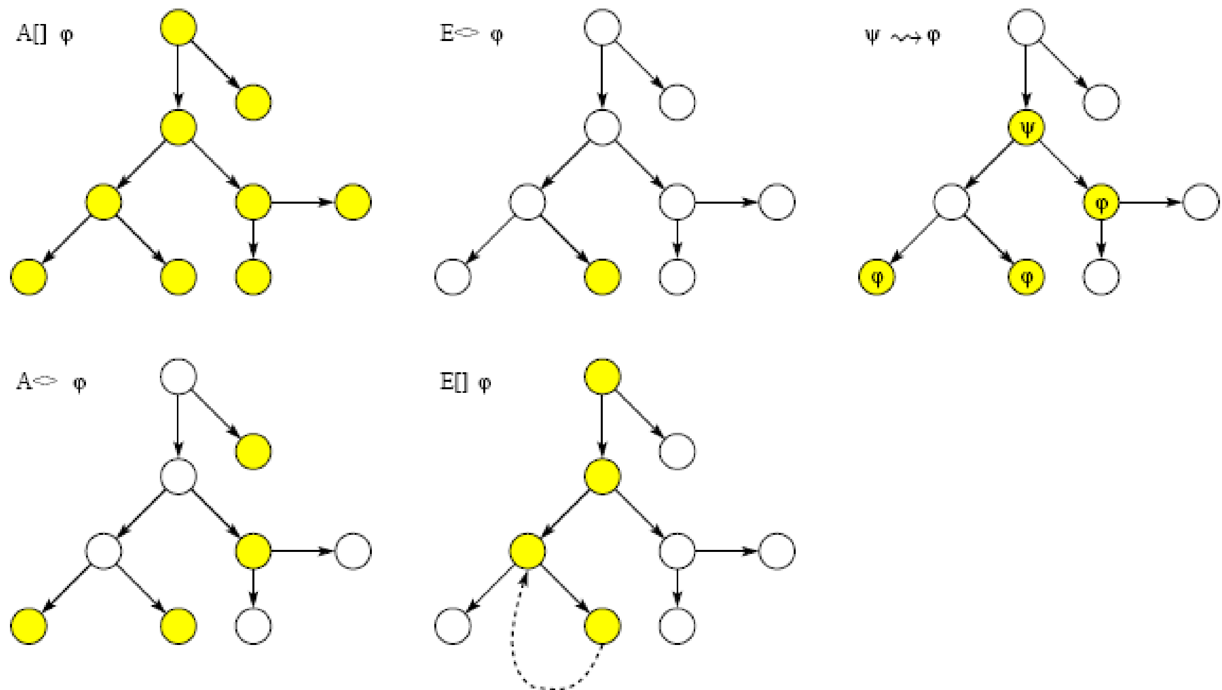


Figura 8: formule TCTL

Il model checker in realtà converte le proprietà illustrate in proprietà contenenti solo l'operatore E:

Name	Property	Equivalent to
Possibly	E<> p	
Invariantly	A[] p	not E<> not p
Potentially always	E[] p	
Eventually	A<> p	not E[] not p
Leads to	p --> q	A[] (p imply A<> q)

Tali proprietà sono verificate tramite un algoritmo di *Reachability analysis*:

```
passed := { }
waiting := { (l0,v0) }
repeat
      begin
            remove (l,v) from waiting
            if (l,v) satisfies β then return success
            else if not (v⊆v') for all (l,v')∈passed then
            begin
                  add (l,v) to passed
                  successor := { (ls,vs) : (l,v)↦(ls,vs) ^ vs≠0 }
                  for all (ls',vs') in successor do
                  put (ls',vs') to waiting
            end
      end
until waiting == { }
return fail
```

L’insieme *waiting* è un insieme di stati (l,v) che devono essere verificati su β, mentre l’insieme *passed* è l’insieme degli stati già verificati. Inizialmente l’insieme *waiting* contiene solamente (l_0,v_0), dove l_0 è il vettore di locazione che contiene gli stati iniziali di tutti gli automi e v_0 è il vincolo iniziale del sistema. L’algoritmo inizia prelevando uno stato dall’insieme *waiting* e termina con successo se questo stato soddisfa β. Altrimenti lo stato è aggiunto all’insieme *passed* se v non è incluso in nessuno dei vincoli di sistema precedentemente verificati. Tutti i successori dello stato sono aggiunti all’insieme *waiting*, sotto la condizione che i vincoli di sistema di questi successori non devono essere vuoti. L’algoritmo si ripete fino a che *waiting* non è vuoto. Se l’insieme si svuota prima che sia stato trovato uno stato che soddisfa β, l’algoritmo termina con fail.

Capitolo 4

Applicazione pratica: rappresentazione e verifica del protocollo DHCP

In questo ultimo capitolo dapprima parleremo di un comune protocollo di configurazione dinamica degli indirizzi IP utilizzato in Internet per andare poi a vedere come è stato modellato in Uppaal e come questo tool, da noi scelto per la verifica dei sistemi real-time, ci consenta di verificarne determinate proprietà.

4.1 DHCP (Dynamic Host Configuration Protocol)

DHCP (Dynamic Host Configuration Protocol) è un protocollo che consente ai dispositivi di rete di ricevere dinamicamente un indirizzo IP per poter connettersi alla rete nella quale si trovano.
Esistono tre modalità di configurazione di un indirizzo IP:
- Manuale: viene assegnato dall'amministratore di rete un indirizzo IP predeterminato per ogni macchina in maniera manuale.
- Automatico: viene assegnato un indirizzo IP alla macchina quando questa si connette per la prima volta alle rete e questo viene mantenuto in maniera permanente per tutte le connessioni future.
- Dinamica: viene assegnato un IP temporaneo alla macchina per un certo periodo di tempo (lease) allo scadere del quale la macchina può riottenere lo stesso indirizzo o un altro.

La modalità che verrà presentata nel nostro studio è quella dinamica.
Questo protocollo fornisce un meccanismo che consente agli host di ricevere delle informazioni di configurazione da parte dei server DHCP utilizzando una rete TCP/IP ed un meccanismo utilizzato dai server DHCP per mantenere in memoria l'associazione (bind) tra indirizzo di rete ed host al quale è stato assegnato.
Tutte le informazioni di configurazione scambiate tra host e server DHCP sono inviate in messaggi UDP singoli. Ogni host è identificato univocamente dal proprio indirizzo fisico, MAC, e dalla rete nella quale si trova ed ogni server DHCP dal proprio indirizzo ip.
Lo scenario gestito dal protocollo consiste in uno o più server DHCP ai quali è assegnato un sottoinsieme dello spazio di indirizzamento che utilizza per assegnare indirizzi agli host che ne facciano richiesta.

4.2 Diagramma temporale

Riportiamo di seguito il diagramma temporale che illustra i vari passi compiuti dal protocollo.

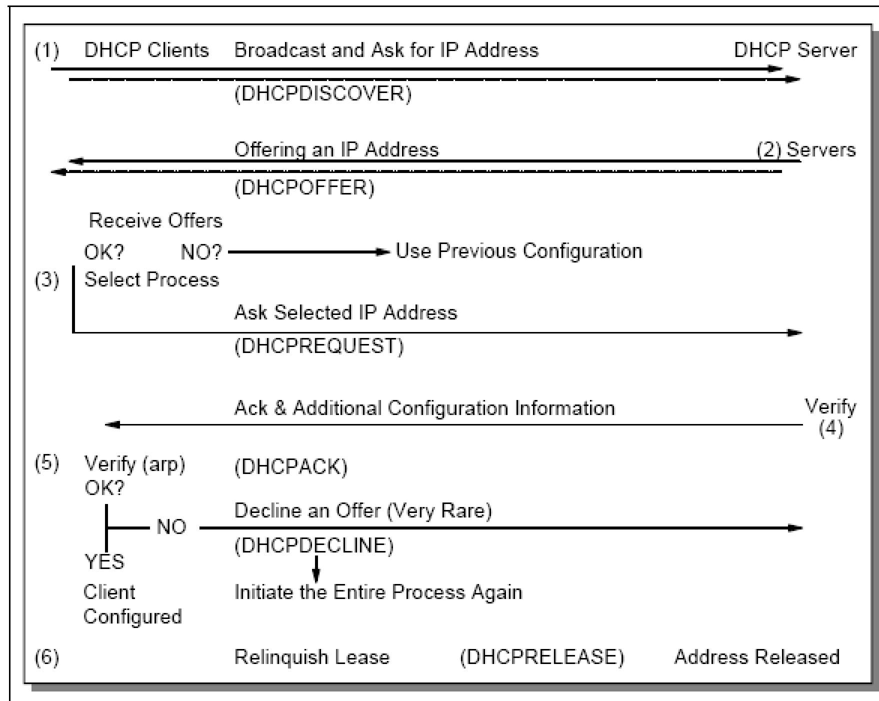


Figura 1: interazione client-server

La figura 1 mostra quali sono i passi fondamentali che un client, nel nostro caso un host, che voglia ottenere un indirizzo IP, deve compiere al fine di poter ottenere un indirizzo valido per potersi connettere nella rete nella quale si trova.

Supponiamo che il client inizialmente non abbia nessun indirizzo valido.

1. Il client invia sulla porta 67 un messaggio UDP DHCPDISCOVER in broadcast a tutti i server DHCP presenti nella rete per richiedere l’assegnazione di un indirizzo.

2. Ogni server DHCP programmato a rispondere, se riceve il messaggio DHCPDISCOVER, invia in risposta un messaggio DHCPOFFER contenente un indirizzo IP ed altri parametri di configurazione e tiene traccia dell’indirizzo IP offerto per evitare che possa offrirlo ad altri eventuali client.

3. Il client, se riceve più offerte, sceglie la prima arrivata e manda in broadcast sulla rete un messaggio DHCPREQUEST specificando l’identificatore del server DHCP dal quale ha accettato l’offerta e l’IP offerto. Il client risponde in broadcast in modo che tutti i server DHCP che avevano fatto l’offerta capiscano di non essere stati accettati e possano liberare l’IP che avevano impegnato per lui.

4. Il server del quale ha accettato l’offerta a questo punto crea un identificatore unico per il lease del client costituito dal MAC del client e dall’IP offerto. Tale lease verrà inviato anche al client nel messaggio di risposta DHCPACK insieme ad altri parametri di configurazione come il tempo che ha a disposizione, il leasetime.

5. Il client riceve DHCPACK, effettua dei controlli preliminari sui parametri e memorizza il lease ed il suo identificatore. Se non rileva errori il client è configurato altrimenti manda un DHCPDECLINE al server e ricomincia la procedura dal punto 1.

6. Se il client riceve un DHCPNACK dal server ritorna al punto 1.

7. Nel momento in cui il client non ha più bisogno dell'indirizzo IP e vuole rilasciarlo invia un DHCPRELEASE al server specificando il proprio lease ovvero MAC ed indirizzo IP associato.

4.3 Diagramma degli stati e delle transizioni

La figura 2 mostra il diagramma degli stati e delle transizioni del client che mostra quali siano gli stati che vengono attraversati dal client durante tutto il suo periodo di vita, ovvero da quando richiede un indirizzo IP a quando rilascia la rete.
Gli archi tratteggiati indicano l'invio di messaggi broadcast, mentre gli archi continui rappresentano l'invio di messaggi unicast.

Figura 2: Diagramma degli stati e delle transizioni di un client DHCP

E' proprio a questo diagramma che, dopo avere fatto le necessarie assunzioni, ci riferiremo nel modellare il protocollo di comunicazione utilizzando Uppaal.
Una volta assunto l'indirizzo IP come descritto nel precedente paragrafo, il lease è regolato da due timer T1 e T2, fissati rispettivamente al 50% e all'87,5% del leasetime. Allo scadere del primo, il client transita nello stato Renewing inviando una DHCPREQUEST al proprio server per ottenere il rinnovo dell'assegnazione. In assenza di risposta, allo scadere del secondo timer, il client transita nello stato Rebinding, inviando DHCPREQUEST in broadcast per mantenere la connessione. Allo scadere del

leasetime, corrispondente in genere ad un'ora, senza aver ricevuto risposta da alcun server, la connessione si interrompe ed il client torna allo stato iniziale.

4.4 Modellazione in UPPAAL

Il diagramma presentato in figura 2 è stato alla base della modellazione sul tool di verifica.

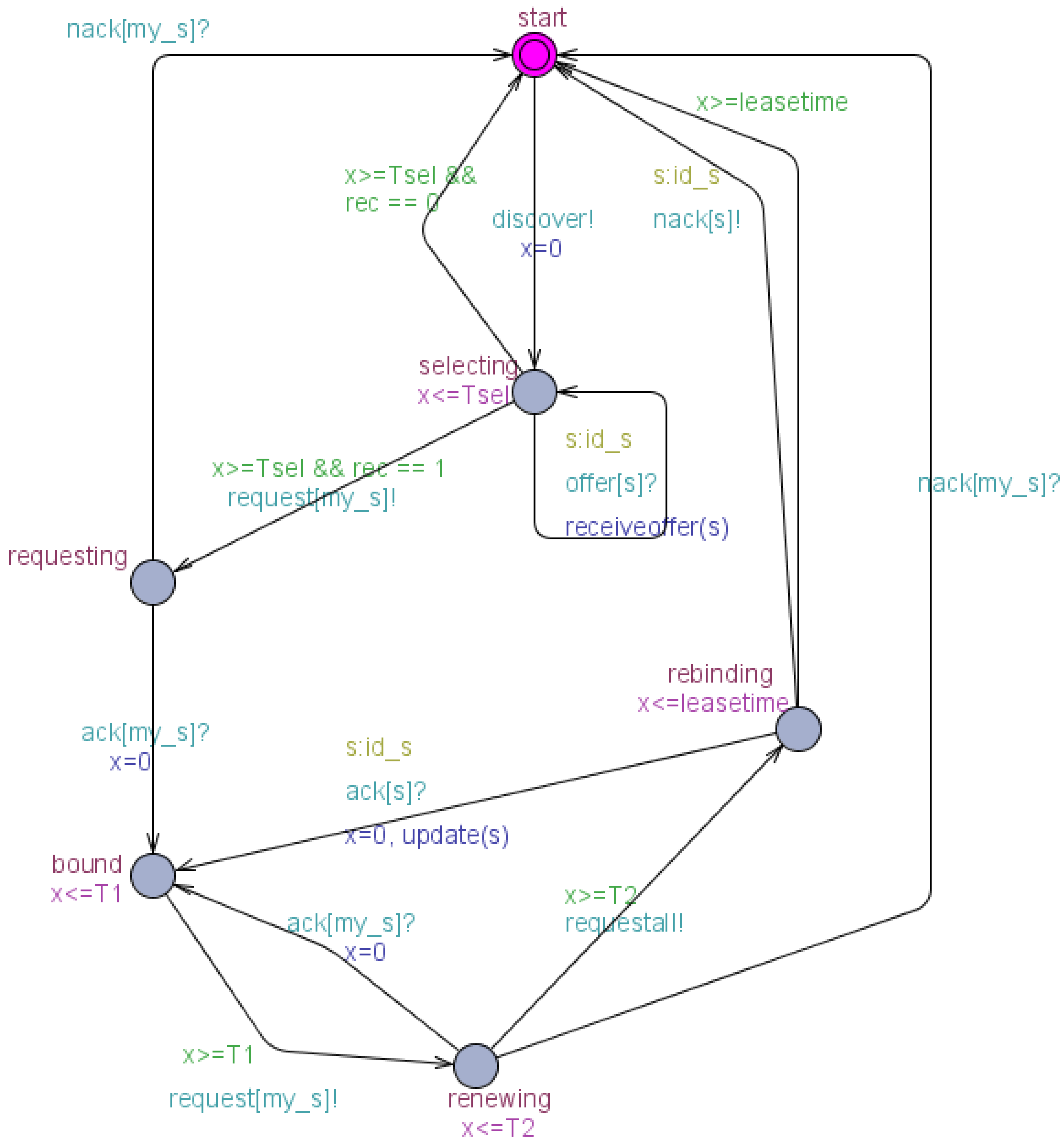


Figura 3: Client

In figura 3 è riportata la modellazione del Client. Per semplicità si suppone che il sistema sia stabile (non necessiti di reboot in seguito a guasti) e il client non sospenda la connessione prima dello scadere del leasetime tramite messaggi di DHCPRELEASE. Con questa premessa, gli stati sono in corrispondenza uno-a-uno con il diagramma in figura 2.

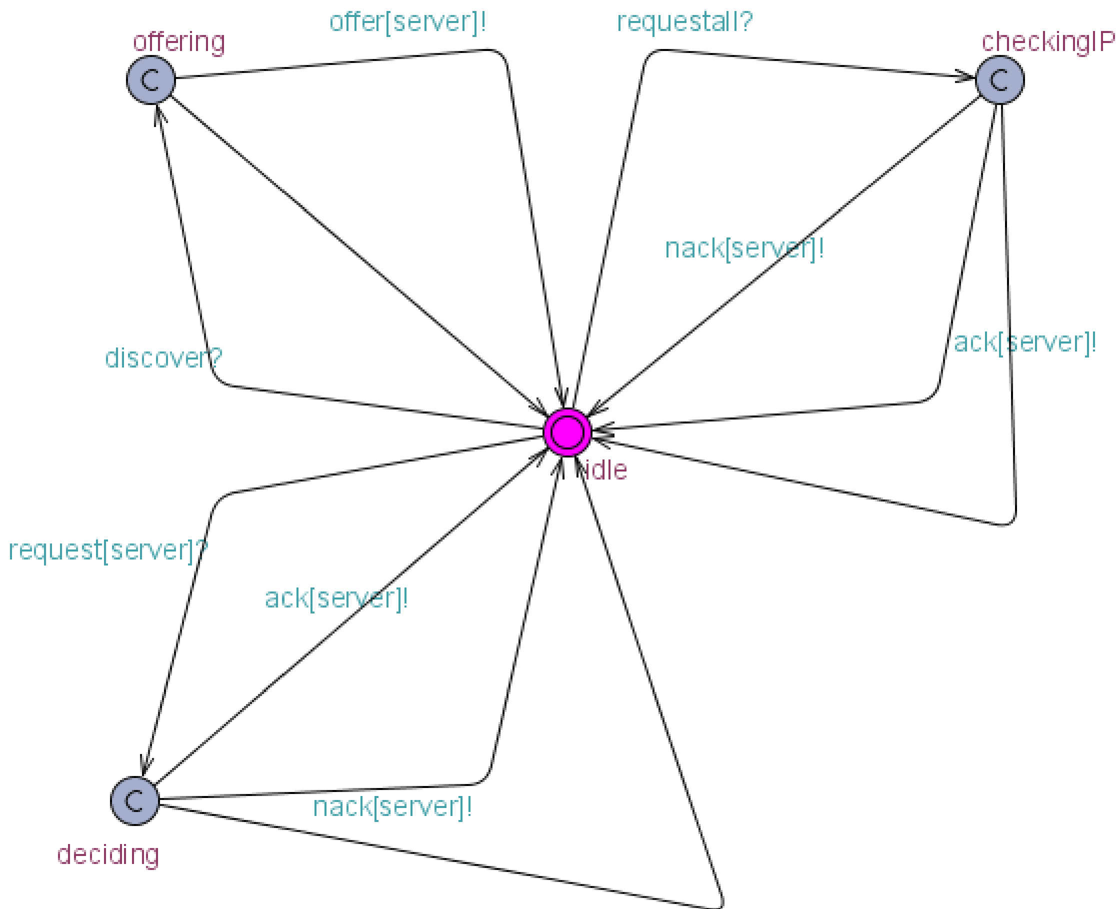


Figura 4: Server

In figura 4 è rappresentato il modello del processo server. Offering, checkingIP e deciding sono stati committed perché il server deve essere sempre nello stato idle e processare le richieste immediatamente. Gli archi senza condizioni sono stati introdotti per simulare la perdita di messaggi UDP in quanto protocollo non affidabile.

4.4.1 Dichiarazioni globali e sincronizzazione

Per rendere leggibile la simulazione, il modello rappresenta un singolo client nell'interazione con N server (in particolare N=3). Il leasetime è fissato a 3600.

La sincronizzazione tra client e server avviene tramite la variabile condivisa id_s, che identifica i server, e i seguenti segnali:

- discover: segnale broadcast che simula l'invio di DHCPDISCOVER ai server.
- offer[id_s]: segnale unicast inviato dal server id_s che simula la proposta di un indirizzo IP disponibile
- request[id_s]: segnale unicast che simula DHCPREQUEST verso il server offerente identificato da id_s.
- ack[id_s]: segnale unicast di conferma da parte del server id_s.
- nack[id_s]: segnale unicast di rifiuto della richiesta proveniente dal server id_s.
- requestall: segnale broadcast per il rebinding che simula i messaggi DHCPREQUEST.

4.4.2 Dichiarazioni e metodi locali

A lato client sono fissati i timer T1 e T2 come da specifica. Inoltre il valore Tsel evita lo stallo nello stato selecting.

Il metodo receiveoffer(id_s) memorizza il server offerente selezionato e setta la variabile booleana rec a 1 per indicare che almeno un'offerta è stata ricevuta e poter passare allo stato requesting; update() aggiorna il server cui il client è connesso dopo aver effettuato il rebinding.

Si è scelto di ipotizzare che gli indirizzi IP di ciascun server siano infiniti e irripetibili, in modo da evitare di costruire complicate strutture dati che non interessano ai fini della verifica delle proprietà del protocollo.

4.5 Simulazione del modello

L'esecuzione nel simulatore del modello sopra presentato, ha evidenziato la presenza di un deadlock nello stato requesting, come si può vedere in figura 5.

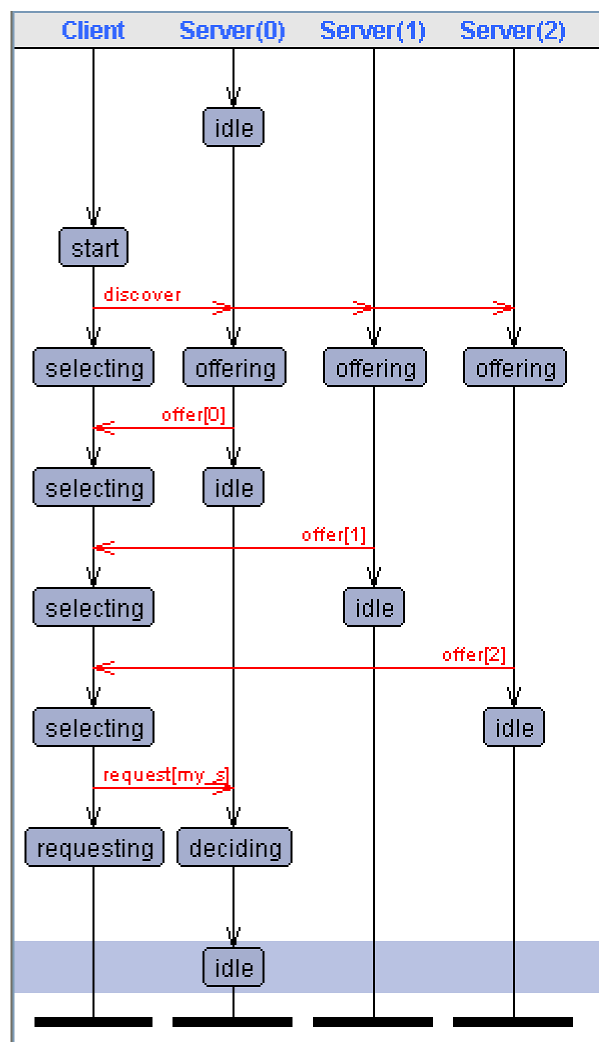


Figura 5: message sequence chart del modello

Il deadlock si verifica nello stato requesting nell'eventualità che il messaggio di acknowledgement del server vada perduto. Per questo motivo si è introdotto presso il client un timer Treq non previsto nelle specifiche che viene utilizzato come invariante dello stato requesting. Se non arriva nessun messaggio di acknowledgement entro tale periodo, si riparte dallo stato iniziale per tentare di nuovo una negoziazione invece di attendere invano. Per questo motivo il client sarà modellato come in figura 6.

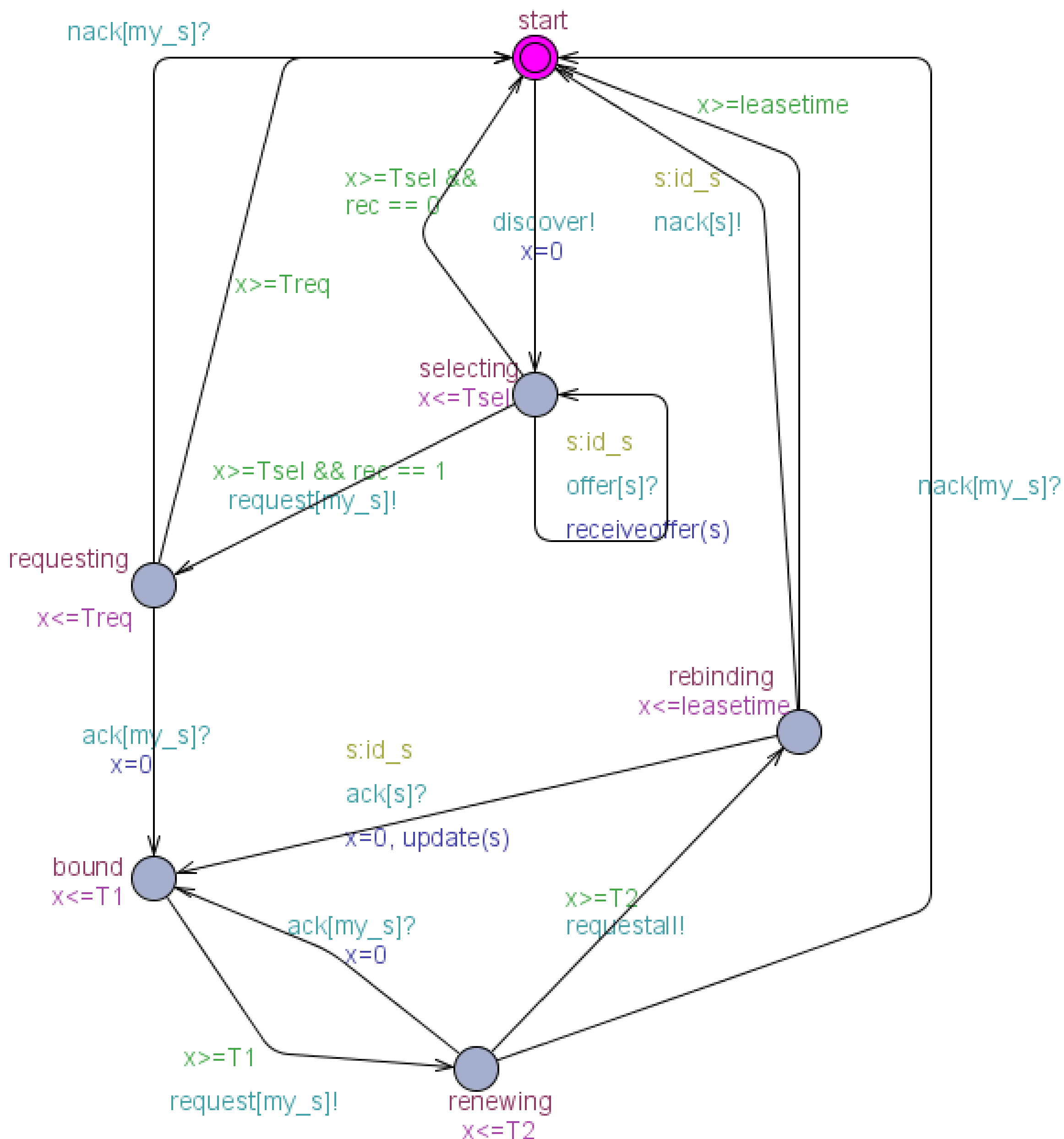


Figura 6: Client nella versione finale

4.6 Model checking

La prima proprietà di interesse del protocollo è quella di raggiungibilità dello stato bound:

E<> Client.bound : esiste un cammino che conduce il client nello stato bound, cioè di instaurare la connessione. Essa risulta soddisfatta.

La seconda proprietà è di liveness:

A<> Client.bound : il client sarà necessariamente condotto nello stato bound. Il modello non soddisfa la proprietà: infatti il protocollo prevede l’uso dei messaggi UDP che possono andare perduti, impedendo al client di dialogare con il server e riuscire a stabilire la connessione. A livello teorico sarebbe necessario prediligere il protocollo TCP per soddisfare la proprietà. In realtà la perdita di pacchetti UDP è in generale molto bassa, perciò i cammini che non garantiscono la connessione sono molto pochi.

Infine, sono state verificate alcune proprietà di safety:

A[] forall (i : id_s) forall (j : id_s) Server(i).deciding && Server(j).deciding imply i == j : non c'è mai più di un server in ogni istante che si trovi a scegliere se confermare l'IP richiesto dal client che aveva ricevuto l'offerta, cioè il client deve averne selezionato uno solo.
A[] !Client.start imply Client.x <= leasetime : il clock non supererà mai il leasetime dopo che il client ha iniziato la procedura, cioè ha lasciato lo stato iniziale.
Entrambe le proprietà sono soddisfatte.

Bibliografia

[1] Rajeev Alur, "A Theory of Timed Automata",in Theoretical Computer Science, http://www.cis.upenn.edu/~alur/Cav99.ps.gz, 1994

[2] Patricia Bouyer, "Timed Automata . From Theory to Implementation" , http://www.lsv.ens-cachan.fr/~bouyer/files/bouyer_chennai.pdf

[3] Johan Bengtsson and Wang Yi, " Timed Automata: Semantics, Algorithms and Tools", in Lecture Notes on Concurrency and Petri Nets. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag, 2004.

[4] Farn Wang, "Formal Verification of Timed Systems:A Survey and Perspective", Proceedings of the IEEE, Vol. 92, Nr. 8, August 2004, pp.1283-1307, IEEE.

[5] Rajeev Alur, Thomas A. Henzinger, "Real-time system=discrete system + clock variables", in Theories and Experiences for Real-Time System Development, AMAST Series in Computing 2 http://www-cad.eecs.berkeley.edu/~tah/Publications/real-time_system=discrete_system+clock_variables.ps 1993

[6] R. Alur, C. Courcoubetis, and D. L. Dill, "Model checking for realtime systems," in Proc. 5th Annu. IEEE Symp. Logic in Computer Science, 1990, pp. 414–425.

[7] Kim G. Larsen, Paul Pettersson, Wang Yi," Diagnostic Model-Checking for Real-Time Systems", http://www.docs.uu.se/docs/rtmv/papers/dimacs95.ps.gz

[8] Rajeev Alur, Bow-Yaw Wang," Next Heuristic for On-the-Fly Model Checking", Lecture Notes In Computer Science; Vol. 1664 Proceedings of the 10th International Conference on Concurrency Theory Pages: 98 – 113 Year of Publication: 1999 http://portal.acm.org/citation.cfm?id=701455

[9] Daniel Kröning – ETH Zürich, "State explosion problem", http://www.inf.ethz.ch/personal/daniekro/classes/251-0247-00/ws2005-2006/slides/smc.pdf

[10] Paul Pettersson,"Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice", a dissertation in Computer Systems for the degree of Doctor of Philosophy. Publicly examined in room X, Uppsala University, 19 February 1999. http://user.it.uu.se/~paupet/thesis.shtml

[11] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL—a tool suite for automatic verification of real-time systems," in Lecture Notes in Computer Science, Hybrid Control Systems. Heidelberg, Germany: Springer-Verlag, 1996, vol. 1066,pp. 232–243.

[12] Gerd Behrmann, Alexandre David and Kim G. Larsen," A Tutorial on Uppaal", in proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04). LNCS 3185. http://www.docstoc.com/docs/2134741/A-Tutorial-on-Uppaal

[13] Kim G. Larsen, Paul Pettersson and Wang Yi, "Uppaal in a Nutshell", in Springer International Journal of Software Tools for Technology Transfer 1(1+2), http://www.it.uu.se/research/group/darts/papers/texts/lpw-sttt97.pdf, 1997

[14] Gerd Behrmann, Kim G. Larsen, Oliver Moller, Alexandre David, Paul Pettersson, Wang Yi, "UPPAAL-Present and Future", http://www.brics.dk/~omoeller/./../papers/cdc01.ps.gz

[15] André Wong, "UPPAAL and SCSI Protocol Example", www.cs.toronto.edu/~chechik/courses97/csc2108/projects/web/1.ps

[16] Help del tool Uppaal, http://www.uppaal.com

[17] Alessandro Artale, "Formal Methods Lecture iv: Computation Tree Logic (CTL)", Faculty of Computer Science – Free University of Bolzano

[18] F. Laroussinie, Ph. Schnoebelen and M. Turuani," On the Expressivity and Complexity of Quantitative Branching-Time Temporal Logics", inTheoretical Computer Science Volume 297 , Issue 1-3 (March 2003) Latin American theoretical informatics Pages: 297 - 315 Year of Publication: 2003 http://www.lsv.ens-cachan.fr/Publis/PAPERS/LST-TCS01.ps

[19] Wang Yi, P. Pettersson and M. Daniels, "Automatic Verification Of Real-Time Communicating Systems by Constraint-Solving".

[20] IBM red book su TCP/IP capitolo 3 "Internetworking protocols" sezione 3.7 "Dynamic Host Configuration Protocol", http://www.redbooks.ibm.com/redbooks/pdfs/gg243376.pdf