TRADUZIONE DI STATE AND TRANSITION DIAGRAMMUML E CORRETTEZZA DEI PROGRAMMI

INDICE

LA GRAMMATICA

INTRODUZIONE

Tla+ è un linguaggio formale di specifica che descrive i sistemi in cui lo stato cambia in passi discreti ed è basato su TLA. Differisce da TLA semplicemente per la presenza dei moduli utilizzati per descrivere le specifiche, ma che illustreremo più avanti.

SINTASSI

Diamo qualche definizione per capire meglio il funzionamento di TLA+.

Uno **stato** è un assegnamento di valori a variabili, mentre la **specifica di un sistema** è la descrizione dell'insieme dei suoi possibili comportamenti che ne rappresentano la corretta esecuzione. Infine un **comportamento**, detto **behaviour**, è una sequenza di stati.

Un tipico comportamento, ovvero una sequenza di stati potrebbe essere:

- **[hr=11] → [hr=12] → [hr=1] → [hr=2] →** …

- **COME SCRIVERE UNA FORMULA LOGICA**

Il **predicato iniziale HCini**: identifica tutti i possibili valori iniziali di hr .

La relazione **next-state (HCnxt)**: descrive come lo stato evolve nel tempo

Es. **HCini = = hr ∈ {1, …, 12}**

HCnxt = = hr' = IF hr ≠ 12 THEN hr+1 ELSE 1

HCnxt è una **formula logica**, chiamata **azione**, che contiene variabili *senza* e *con apice (il 'prima' e il 'dopo')*.

Una azione può essere vera o falsa rispetto a un passo.

Un **passo** detto **step** è una coppia di stati adiacenti

- **Esempio: [hr=1] → [hr=2]**

La sintassi di TLA + è costruita utilizzando gli operatori della logica proposizionale, il linguaggio degli insiemi, gli operatori modali e quantificatori con dichiarazione di range.

- ## OPERATORI

¬ NOT

V OR

^ AND

⇒ IMPLICAZIONE LOGICA

≡ EQUIVALENZA LOGICA

- ## OPERATORI TEMPORALI

ALWAYS :

Se f è una formula logica, la formula temporale F asserisce che **la formula F è sempre vera per ogni stato di ogni comportamento.**

HCini $\wedge$ HCnxt

Dato un comportamento, la formula temporale puo' essere vera o falsa rispetto ad esso. Dunque la formula denota tutti i comportamenti che la soddisfano, cioè quelli in cui:

- il primo stato soddisfa HCini
- ogni passo soddisfa HCnxt

Un **teorema** è una formula temporale soddisfatta da ogni comportamento dell'universo Esempio. La formula:

'HC => HCini'

è soddisfatta da ogni comportamento.

EVENTUALLY ◊:

Se f è una formula logica, la formula temporale ◊ F asserisce che **la formula F eventualmente è vera.**

F =¬ F

◊ F: Eventualmente F è sempre vera.

- ## QUANTIFICATORI

QUANTIFICATORE UNIVERSALE: $\forall$

QUANTIFICATORE ESISTENZIALE: $\exists$

LA SPECIFICA IN TLA+

La specifica in TLA+ differentemente da TLA è descritta attraverso l'uso di una struttura detta **modulo.**

Vediamone un esempio:

```
---------------- MODULE HourClock ----------------
EXTENDS Naturals
VARIABLE hr
HCini == hr \in (1 .. 12)
HCnxt == hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC == HCini /\ [][HCnxt]_hr
------------------------------------------------------
THEOREM HC => []HCini
```

Dove:

- **EXTENDS** descrive i moduli che vogliamo utilizzare, in questo caso abbiamo importato il modulo dei Naturali.
- **VARIABLE** definisce le variabili che andremo ad utilizzare nel modulo.
- **AZIONI** descrivono in che modo evolve il sistema. Il predicato iniziale: identifica tutti i possibili valori iniziali di hr .
- **THEOREM** indica una formula che deve essere vera in questo contesto.

  Formalmente il modulo non identifica quale definizione è la specifica e quali sono le ausiliarie. Questo viene stabilito in un file a parte **.cfg**

  Es. di configuration file:

```
CONSTANTS
        SIZE = 4
        WHITE = "WHITE"
        BLACK = "BLACK"

SPECIFICATION Spec

INVARIANT TypeInvariant \* Prop1
```

  Altre definizioni utili:

  Una **funzione di stato** è una espressione ordinaria che può contenere variabili e costanti.

  Un **predicato di stato** è una funzione di stato a valore booleano.

  Un **invariante** *Inv* di una specifica *Spec* è un predicato di stato tale che:

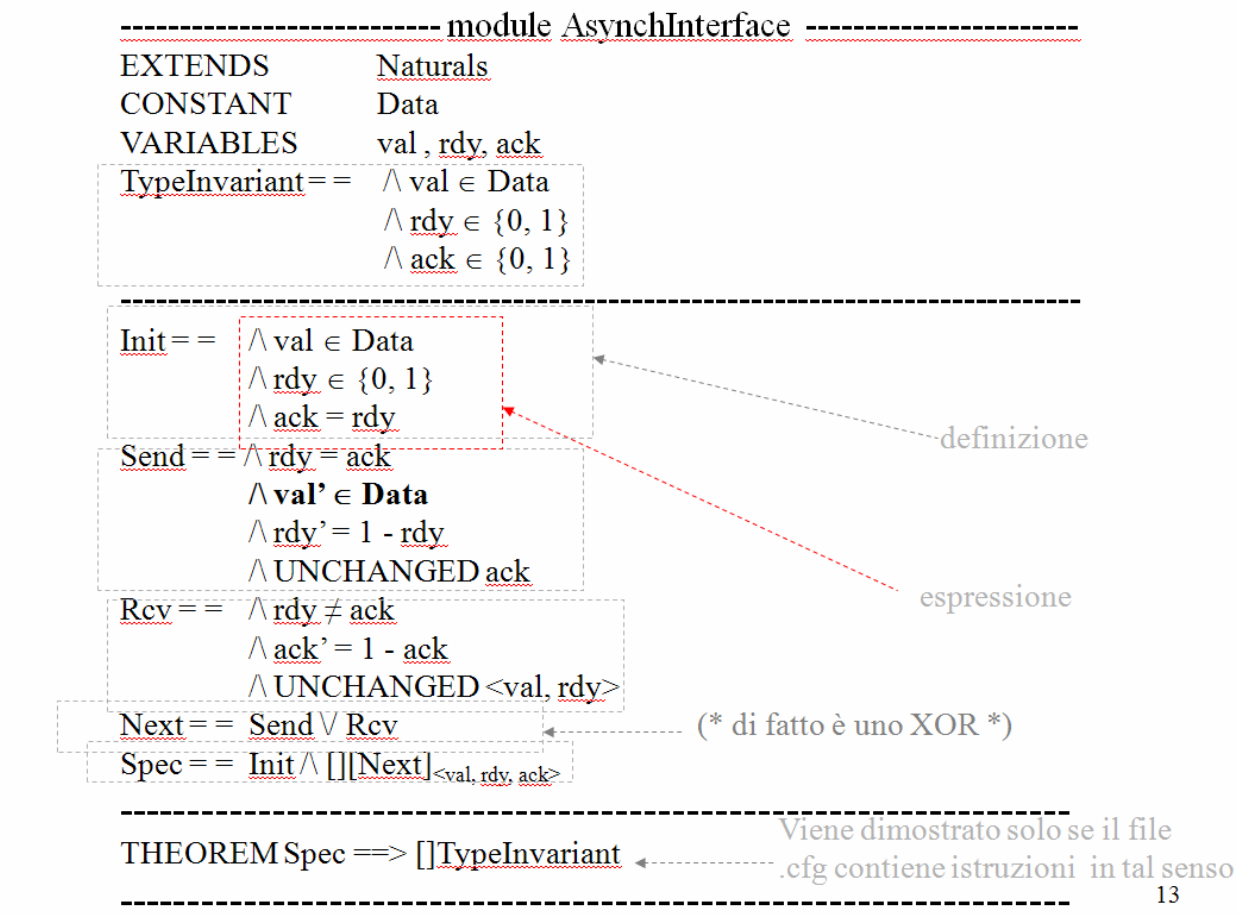
  Spec ⇒ *Inv* è un teorema.

  Una variabile *v* ha tipo *T* in una specifica *Spec* sse $v \in T$ è un *invariante* di *Spec*.

  – *hr* ha tipo 1..12 nella specifica *HC vista precedentemente (non in ogni comportamento dell'universo).*

  – Di fatto, TLA+ è un linguaggio senza tipi.

Continuiamo con i nostri esempi che sono il modo migliore per capire la struttura di un modulo TLA+.



13

- # I RECORD

In TLA+ possiamo utilizzare il concetto di **RECORD** dei linguaggi di programmazione.

Un insieme di record può essere descritto esplicitamente così:

chan Є [val: Data, rdy:{0,1}, ack:{0,1}]

L'ordine dei campi è irrilevante.

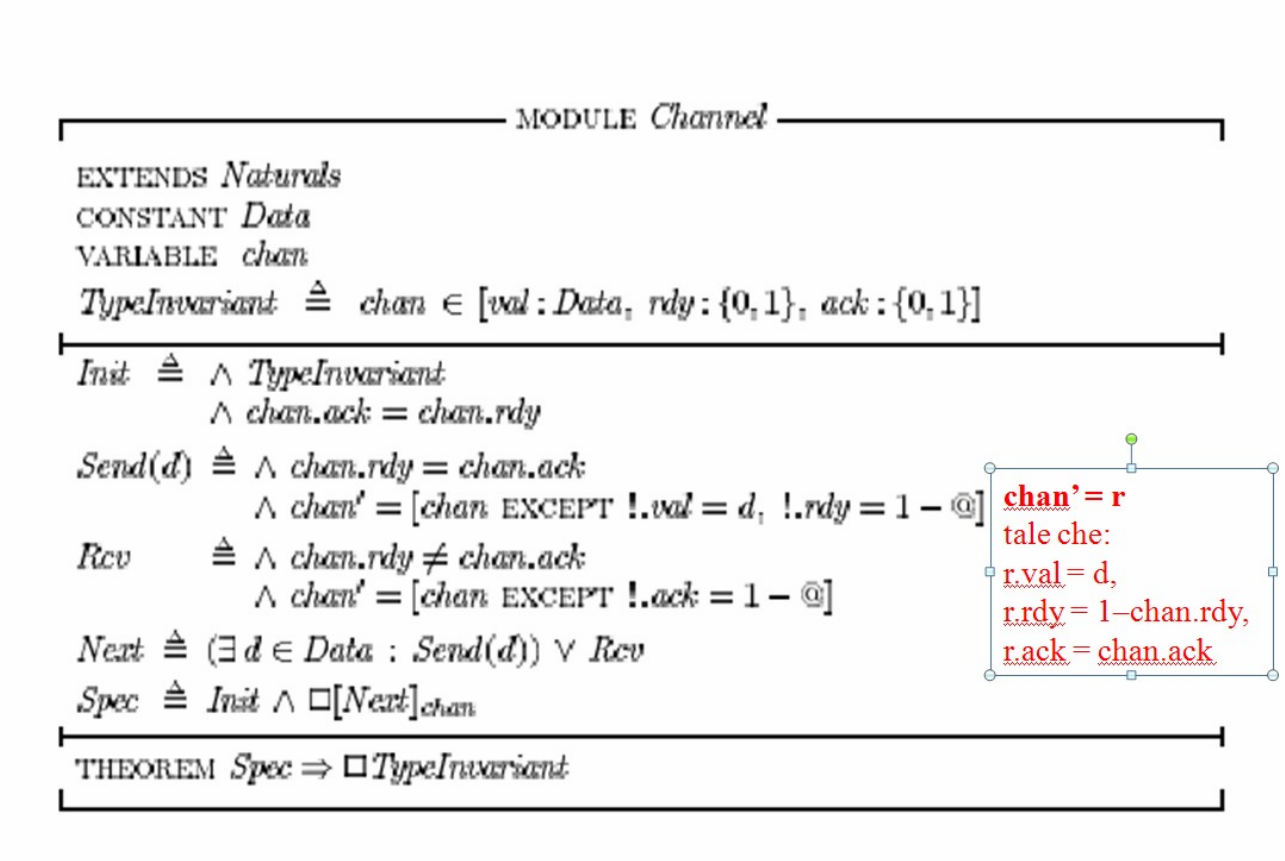
Quindi un possibile elemento **chan** potrebbe essere:

[val→d, rdy→1, ack→0]

Ed i campi possono essere estratti mediante l'istruzione:

chan.val, chan.rdy, chan.ack

Esempio:



Send, ***Init*** e ***Spec*** sono identificatori.

Send(e) denota un'espressione; per ogni espressione e , e può essere usato come sotto-espressione di espressioni complesse.

Es. Send(-5) denota

^ chan.rdy = chan.ack

^ chan' = [chan except !.val = -5, !.rdy = 1 - @]

Send è anche un operatore che prende un solo argomento.

Init e ***Spec*** si possono vedere come operatori senza argomenti.

La definizione generale di un operatore a più argomenti è:

- *Id(p1, ..., pn) = = exp* (*Id*, *p1*, …, *pn* sono identificatori).

- **DICHIARAZIONE, DEFINIZIONE, DOMINIO(SCOPE)**

  Ogni *simbolo* usato in un modulo deve essere
  - dichiarato (CONSTANT *Data*, VARIABLE *hr*),
  - definito (*Rcv* ==…) o
  - predefinito (EXTENDS Naturals, con '+').

- Lo ***scope*** (*dominio di definizione*) di una *dichiarazione* di CONSTANT o VARIABLE, o di una *definizione*, è la parte del modulo che la segue.
- La definizione *Id(p1, ..., pn) = = exp* introduce implicitamente le dichiarazioni delle variabili *p1, ..., pn*, il cui scope è *exp*.
- L'espressione $\exists v \in S : exp$ dichiara una var. *v* il cui scope è *exp* (non l'espressione *S*).

- Due identificatori uguali devono avere dominii disgiunti

- **TUPLE**

Possiamo usare anche il concetto di **TUPLA** che viene denotato con:

$$\begin{aligned}
&\langle 1, 2, 3\rangle &&\in Nat \times Nat \times Nat\\
&\langle\langle 1, 2\rangle, 3\rangle &&\in (Nat \times Nat) \times Nat\\
&\langle 1, \langle 2, 3\rangle\rangle &&\in Nat \times (Nat \times Nat)
\end{aligned}$$

Le 3 tuple non sono uguali ed x non è un operatore associativo.

TLA+ definisce una **STRINGA** come una tupla di caratteri.

- **ISTANZIAZIONE=SOSTITUZIONE**

MODULE *InnerFIFO*

EXTENDS *Naturals*, *Sequences*
CONSTANT *Message*
VARIABLES *in*, *out*, *q*
$InChan \triangleq$ INSTANCE *Channel* WITH *Data* $\leftarrow$ *Message*, *chan* $\leftarrow$ *in*
$OutChan \triangleq$ INSTANCE *Channel* WITH *Data* $\leftarrow$ *Message*, *chan* $\leftarrow$ *out*

$$\begin{aligned}
Init \triangleq\ & \land\ InChan!Init\\
& \land\ OutChan!Init\\
& \land\ q = \langle\rangle
\end{aligned}$$

$$\begin{aligned}
TypeInvariant \triangleq\ & \land\ InChan!TypeInvariant\\
& \land\ OutChan!TypeInvariant\\
& \land\ q \in Seq(Message)
\end{aligned}$$

MODULE *InnerFIFO*

EXTENDS *Naturals*, *Sequences*
CONSTANT *Message*
VARIABLES *in*, *out*, *q*
$InChan \triangleq$ INSTANCE *Channel* WITH *Data* $\leftarrow$ *Message*, *chan* $\leftarrow$ *in*
$OutChan \triangleq$ INSTANCE *Channel* WITH *Data* $\leftarrow$ *Message*, *chan* $\leftarrow$ *out*

$$\begin{aligned}
Init \triangleq\ & \land\ InChan!Init\\
& \land\ OutChan!Init\\
& \land\ q = \langle\rangle
\end{aligned}$$

$$\begin{aligned}
TypeInvariant \triangleq\ & \land\ InChan!TypeInvariant\\
& \land\ OutChan!TypeInvariant\\
& \land\ q \in Seq(Message)
\end{aligned}$$

La definizione nel modulo *InnerFIFO*:

$InChan \triangleq$ INSTANCE *Channel* WITH *Data* $\leftarrow$ *Message*, *chan* $\leftarrow$ *in*

definisce InChan come la formula ottenuta da quella ‘vera’ def. Sostituendo la costante Message a

Data,e la variabile in/chan. TUTTI I PARAMETRI del modulo Channel vanno coperti. Dunque la vera def. di InChan!Next è solo in termini di:

- Operatori standard TLA+
- I parametri *Message* e *in* del modulo *InnerFIFO*.

- **Sostituzioni implicite e istanziazione senza *renaming***

  $InChan \triangleq \text{INSTANCE } Channel \text{ WITH } chan \leftarrow in$

  Questa definizione contiene implicitamente anche la sostituzione *Data* ← *Data*, perché tutti i parametri di *Channel* devono essere coperti.

  Il simbolo *Data* deve essere noto (dichiarato o definito) nello *scope* in cui appare la INSTANCE.

  $\text{INSTANCE } Channel \text{ WITH } Data \leftarrow D, chan \leftarrow x$

  Quando serve una sola istanza di un modulo (ex: un solo *Channel*), non occorre introdurre un nuovo nome di variabile per essa.

  Dopo questa istanziazione, posso usare ad esempio il simbolo *Rcv (non Channel!Rcv)*, il cui significato è quello definito nel MODULE *Channel*, eccetto che *Data* ← *D* e *chan* ← *x.*

  Se poi mancano anche queste ultime sostituzioni (le ho solo implicite), è come avere *EXTENDS Channel.*

La **ASSUME***,* che puo' riferirsi solo a costanti, non ha alcun effetto sulle definizioni del modulo, ma puo' essere usata come ipotesi per dimostrare teoremi (THEOREM) del modulo.

La Assume viene automaticamente verificata da TLC.

ALTRI OPERATORI

- **ExistsIn (S, x: x + z > y)**

  Ci indica che: ∃ x Є S : x + z > y

- **Funzioni ricorsive.**

  Nei moduli TLA+ possono essere descritte anche le funzioni ricorsive, vediamone un esempio.

  Questa definizione non è valida, in quanto ciclica:

  fact = = [n Є Nat |-> IF n = 0 THEN 1 ELSE n * fact[n-1]]

  Tuttavia TLA+ ammette la definizione:

  fact [n Є Nat] = = IF n = 0 THEN 1 ELSE n * fact[n-1]

  Come abbreviazione di

  fact = = CHOOSE f:

f = [n Є Nat |-> IF n = 0 THEN 1 ELSE n * f[n-1]]

…che è come scrivere x = = CHOOSE y: y+2 = y*2

- **Choose x ∈ S: p**

Restituisce un valore v di S tale che p, sostituendo v ad x, è vera.

- **If/Then/Else e Case**

TLA+ possiede anche due costrutti condizionali:

IF p THEN e1 ELSE e2 dove l'espressione è uguale a e1 se p è vero, mentre se p è falso è uguale a e2 e un altro costrutto **CASE** che semplifica l'espressione precedente sostituendo If/Then/Else con un unico operatore.

CASE ha 2 forme generali:

$$\text{CASE } p_1 \to e_1 \ \Box \ \ldots \ \Box \ p_n \to e_n$$

$$\text{CASE } p_1 \to e_1 \ \Box \ \ldots \ \Box \ p_n \to e_n \ \Box \ \text{OTHER} \to e$$

$$\text{CASE } n \geq 0 \to e_1 \ \Box \ n \leq 0 \to e_2$$

Es.
L'espressione:

$$\text{CASE } n \geq 0 \to e_1 \ \Box \ n \leq 0 \to e_2$$

È uguale a e1 se n>0 è vero, è uguale a e2 se n<0 è vero e uguale a uno tra e1 o e2 se n=0 è vero.

If/Then/Else e Case sono definiti come segue in termini di **CHOOSE**:

$$\text{IF } p \text{ THEN } e_1 \text{ ELSE } e_2 \triangleq$$
$$\quad \text{CHOOSE } v : (p \Rightarrow (v = e_1)) \wedge (\neg p \Rightarrow (v = e_2))$$

$$\text{CASE } p_1 \to e_1 \ \Box \ \ldots \ \Box \ p_n \to e_n \triangleq$$
$$\quad \text{CHOOSE } v : (p_1 \wedge (v = e_1)) \vee \ldots \vee (p_n \wedge (v = e_n))$$

$$\text{CASE } p_1 \to e_1 \ \Box \ \ldots \ \Box \ p_n \to e_n \ \Box \ \text{OTHER} \to e \triangleq$$
$$\quad \text{CASE } p_1 \to e_1 \ \Box \ \ldots \ \Box \ p_n \to e_n \ \Box \ \neg(p_1 \vee \ldots \vee p_n) \to e$$

- **Let/In**

$$\text{LET} \quad d \triangleq f \quad \text{IN} \quad e$$

Uguaglia e nel contesto della definizione $d \triangleq f$

Es.

$$\text{LET} \quad sq(i) \triangleq i * i \quad \text{IN} \quad sq(1) + sq(2) + sq(3)$$

è uguale a 1*1+2*2+3*3 che è uguale a 14.

TRADUZIONE DI STATE AND TRANSIOTION DIAGRAMM

SISTEMI ASINCRONI

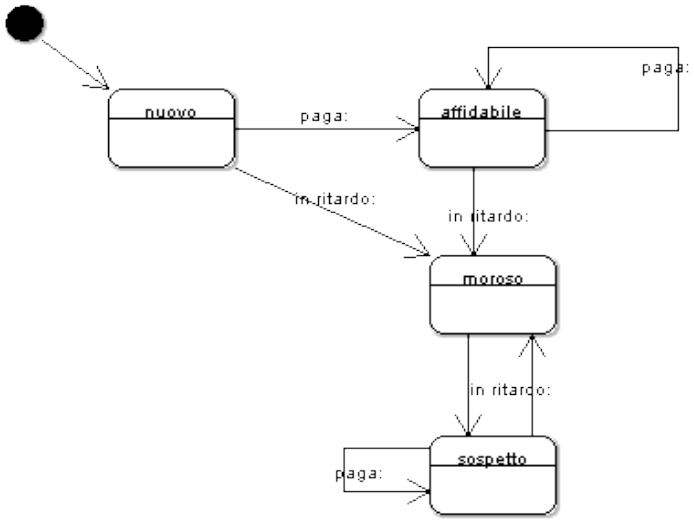
In questo paragrafo ci proponiamo di compiere la traduzione di un diagramma degli stati e delle transizioni UML nella logica temporale di Lamport.
Ricordiamo che un diagramma di sì fatta specie è costituito da due elementi essenziali: **stati** e **transizioni;** una transizione risponde alla forma:

Evento[Condizione]/Azione

ove l'evento indica un accadimento esterno che attiva la transizione, la condizione è una "guardia" che consente la visibilità di un evento, l'azione un comportamento atomico che il sistema deve eseguire.
In questo scritto ci limiteremo a proporre traduzioni di diagrammi privi di condizioni per questioni di *brevitas*, ma la loro inclusione nella metodologia generale è banale, non comportando al lettore che volesse usufruirne eccessivo sforzo concettuale.
La riduzione di un simile formalismo in TLA+ prevede pertanto che ogni elemento sintattico di UML possieda un corrispettivo sintattico e semanticamente equivalente nel linguaggio di traduzione.
Prima di proseguire è bene affermare che l'intuizione sulla metodologia di traduzione non è stata da noi creata ex-novo bensì ricalca, con i dovuti accertamenti, quella mostrata da *Maurizio Lenzerini* e *Marco Cadoli* nel corso di *Progettazione del Software I,* tenuto per il corso di laurea in *Ingegneria Informatica,* per gli stessi diagrammi ma verso il linguaggio Java.
Abbiamo visto che la logica temporale di Lamport è costruita partendo da una base proposizionale, una insiemistica e fondendo a queste una base modale dalla quale deriva i connettivi temporali; certi di ciò, rappresentiamo un diagramma per mezzo di tre moduli TLA+, uno che rappresenti il sistema di stati e transizioni, uno sfruttato come un contenitore di azioni, ed uno servente degli altri, che ne avvia l'esecuzione. In questo quadro i costituenti sintattici vengono rappresentati:

- ✔ Stati: variabile di stato appartente ad un insieme di cardinalità pari al numero degli stati del diagramma UML; per comodità assumeremo sempre che sia un intero;
- ✔ Eventi: questi divengono le azioni che attivano le transizioni in TLA+;
- ✔ Condizioni: variabile booleana di stato;
- ✔ Azioni: azioni appartenenti allo stesso modulo che vengono attivate dagli eventi.

Un accorgimento particolare è da usare nella stesura di un evento: difatti il diagramma deve essere deterministico, perciò è bene discriminare, per mezzo del costrutto **CASE** quando una transizione è applicabile.
Sì da ben comprendere come si compie una traduzione è bene procedere nel mostrare un esempio concreto; in questo primo frangente mostreremo un diagramma costituito da soli eventi, procedendo per gradi.



In loco non riconosciamo la presenza né di operazioni né di condizioni, perciò è sufficiente la stesura di un solo modulo:

clienteBancario

EXTENDS Naturals;

VARIABLES stato;

(mapping degli stati:*
stato=0 →nuovo
stato=1 →affidabile
stato=2 →moroso
stato=3 →sospetto
**)*

TYPE_INVARIANTS $stato \in (0,1,2,3)$

INIT $\triangleq$ *stato=0*

paga $\triangleq$ *stato'=CASE*
stato=0 →1 □
stato=1 →=1 □
stato=2 →3 □
stato=3 →3

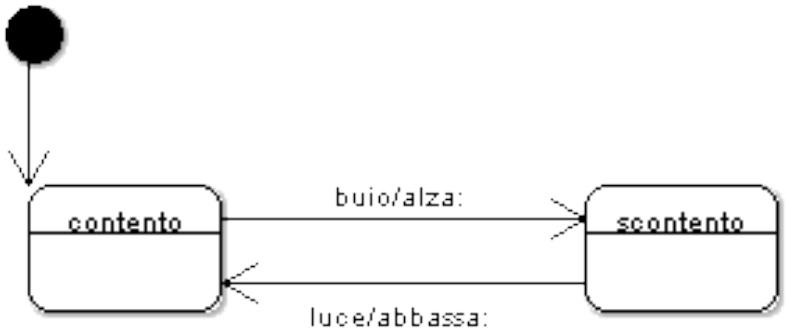
ritardo $\triangleq$ *stato'=CASE*
stato=0 →2 □
stato=1 →2 □
stato=3 →2

Next $\triangleq$ $paga \vee ritardo$

Spec $\triangleq$ $INIT \wedge \lfloor Next \rfloor_{<stato>}$

THEOREM Spec $\Rightarrow$ *TYPE_INVARIANS*

Riconosciamo in quanto scritto due sole azioni, *paga* e *ritardo*, entrambe derivanti dalla traduzione degli eventi del diagramma UML; le azioni *Next* e *Spec*, che potremmo considerare come standard hanno esclusivo scopo quello di, il primo, raccogliere tutte le azioni attraverso le quali il nostro sistema può evolvere, e, il secondo, unico a poter possedere operatori temporali non solo di primalità, descrive l'evolversi del sistema nel tempo.
Per ciò che riguarda la verifica di proprietà sul diagramma ci torneremo in seguito, basti sapere che prediligeremo il ricorso a strumenti automatici, piuttosto che la verifica manuale come proposto dallo stesso autore.
Introduciamo a questo punto una complicazione alla nostra traduzione, le azioni: tradurremo un diagramma più ricco sintatticamente di quello già realizzato.



Pur essendo di complessità inferiore al precedente in relazione alla cardinalità degli stati, questo diagramma si propone come più complesso dal punto di vista sintattico giacché propone l'esecuzione di azioni atomiche durante la transizione di stato.
Il formalismo di TLA+ ci permette di rappresentare le transizioni, sicché nella descrizione di queste è possibile applicare una delle azioni. Le azioni UML saranno a loro volta tradotte come delle azioni TLA ma che non posseggono vita propria, pertanto non faranno parte della specifica o del predicato Next, bensì verranno invocate solo dalle azioni-eventi.
Per evitare incomprensioni o sentimenti di confusione nel lettore proponiamo di seguito la traduzione effettiva del diagramma.
Anche in questo caso è sufficiente la scrittura di un solo modulo, quanto utilizzato nell'esempio precedente è ovviamente ancora valido.
In modo da disattivare definitivamente le azioni UML ricorriamo ad un trucco combinatorio, ovvero imponiamo l'attivazione dell'azione TLA+ esclusivamente al verificarsi di una condizione, condizione attivata dalla transizione.

Bambino

EXTENDS: Naturals

VARIABLES: stato, cond

(mapping degli stati:*
stato=0 →contento
stato=1 →scontento
**)*

$TYPE\_INVARIANTS \triangleq \quad \wedge stato \in \{0,1\}$
$\wedge cond \in \{0,1\}$

$INIT \triangleq \quad \wedge stato = 0$
$\wedge cond = 0$

(AZIONI *)*

$alza \triangleq \quad \wedge cond' = CASE\ cond=0 \rightarrow 1 \square TRUE \rightarrow cond$

$abbassa \triangleq \quad \wedge cond' = CASE\ cond=1 \rightarrow 0 \square TRUE \rightarrow cond$

(EVENTI *)*

$buio \triangleq \quad \wedge stato' = CASE\ \ stato=0 \rightarrow 1 \ \square\ TRUE \rightarrow stato$
$\wedge alza$

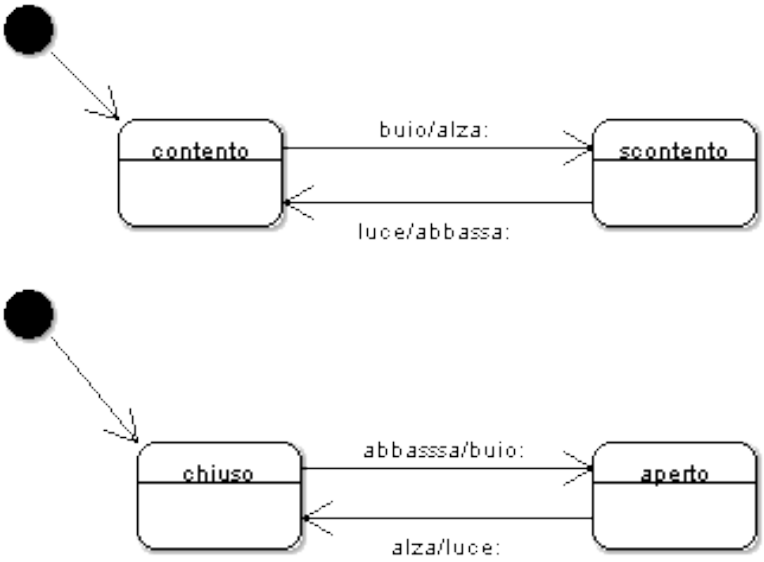
$luce \triangleq \quad \wedge stato' = CASE\ \ stato=1 \rightarrow 0 \ \square\ TRUE \rightarrow stato$
$\wedge abbassa$

$Next: \quad buio \vee luce$

$Spec: \quad INIT \ \& \ \lfloor Next \rfloor_{<stato>}$

SISTEMI SINCRONI

Per ciò che riguarda la traduzione di sistemi tra loro sincroni è leggermente più complicato, è necessario difatti ricorrere ad un modulo per ogni sistema ai quali si assomma un modulo addizionale di coordinamento.
Sì da evitare confusione mostriamo per mezzo di un esempio quali siano le nostre intenzioni; ci riferiremo a quanto già visto in precedenza, ovvero al diagramma del bambino che giocava colla luce, colla variante che scegliamo di dotare di "vita propria" anche l'interruttore, cioè



I mezzi messi a nostra disposizione ci impediscono di poter sfruttare la medesima metodologia di confronto tra sistemi sincroni utilizzata con strumenti quali NUSMV, difatti TLA+ non permette l'interazione congiunta tra i moduli, cioè non è possibile ricorrere al paradigma progettuale *Observer-Observable* in cui un sistema invochi i metodi dell'altro e viceversa, TLA+ non ammette cicli di questo genere.
Sì da scavalcare tale impedimento, abbiamo deciso di modellare nonsì il comportamento di un due sistemi che interagiscono tra loro, bensì quello di un unico sistema che qui in questa sede definiremo **sistema prodotto**.
Il sistema prodotto non fa altro che appianare la differenza tra eventi ed azioni, si propongono perciò nella sequenzialità che consente di evolvere il sistema, in questo caso, assunti gli stati iniziali, osserviamo in loco la sequenza:

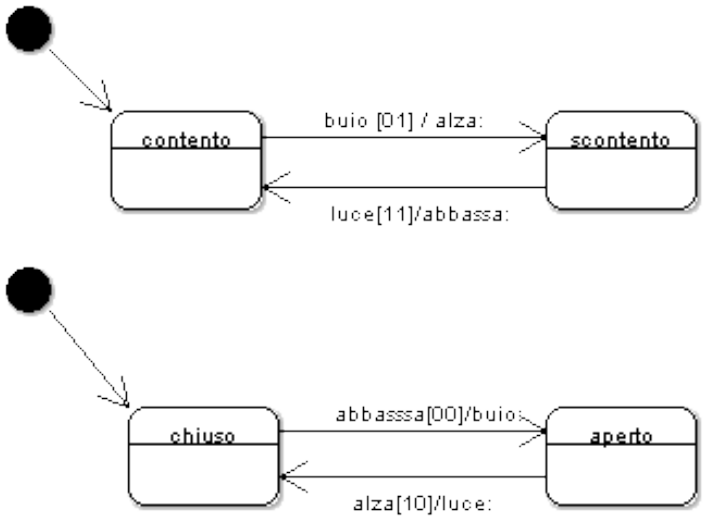
$$abbassa \rightarrow buio \rightarrow alza \rightarrow luce\ .$$

A questo punto codifichiamo in binario, per mezzo di due bit, la posizione di ogni azione/evento nella sequenza sì da avere la corrispondenza biunivoca che segue:

$$abbassa \rightarrow 00$$
$$buio \rightarrow 01$$
$$alza \rightarrow 10$$
$$luce \rightarrow 11.$$

La codifica eseguita non è a se stante ma possiede un'utilità propria difatti questa verrà utilizzata come guardia nell'attivazione degli eventi-azioni: essendo le azioni UML codificate come azioni TLA+, queste potrebbero essere in realtà attivate in qualsiasi momento, ovvero si potrebbe decidere di attivare buio, luce in un modulo Bambino senza che né *abbassa* né *alza* siano mai state eseguite nel modulo interruttore. Per ovviare a tale problema la codifica di ogni azione diventa una *guardia* cioè una condizione che determina l'esecuzione o meno dell'azione in questione.
L'alterazione della guardia è demandata proprio all'esecuzione di un'azione UML contestualmente al verificarsi di un evento: ad esempio l'azione *abbassa* permette l'esecuzione di *buio* nel modulo

locale, la quale determina il cambiamento delle variabili, da 00 a 01, e permette, nell'istante di tempo successivo, l'esecuzione e sola consumazione dell'evento *buio* nel modulo concorrente.
Sì da permettere di redigere in maniera non errata i moduli TLA che realizzano i due sistemi, riscriviamo i diagrammi UML dei sistemi integrati colla codifica, codifica che quivi è visualizzata sotto forma di condizioni:



A questo punto realizziamo in TLA+ due moduli che rispettivamente codifichino i due sistemi, nella maniera già vista, ed un modulo di congiunzione che permetta la condivisione delle variabili di controllo:

---- Bambino ----

EXTENDS Naturals
VARIABLES cond1, cond2,stato

(* mapping degli stati:
stato=0 → contento
stato=1 → scontento
*)

TYPE_INVARIANTS ≙ stato ∈ {0,1}
∧ cond1 ∈ {0,1}
∧cond2 ∈ {0,1}

INIT≙ stato=0
∧cond1=0
∧ cond2=0

(* AZIONI *)

LOCAL alza ≙ ∧ (cond1'=CASE (stato=0 ∧ cond1=0 ∧ cond2=1) →1 €
TRUE → cond1)
∧ (cond2'=CASE (stato=0 ∧ cond1=0∧cond2=1) →0 €
TRUE → cond2)

LOCAL abbassa ≙ ∧(cond1'=cond1)
∧(cond2'=CASE (stato=1 ∧ cond1=1 ∧ cond2=1) →1 €

TRUE → cond2)

(* EVENTI*)

buio ≜ ∧(stato'= CASE (stato=0 ∧ cond1=0 ∧ cond2=1)→1 ∈TRUE → stato)
∧ alza

luce ≜ ∧(stato'= CASE (stato=1 ∧ cond1=1 ∧ cond2=1)→ 0 ∈TRUE → stato)
∧ abbassa

Next ≜ ∨ buio ∨ luce

Spec ≜ INIT ∧∈⌊Next⌋$_{<stato, cond1, cond2>}$
THEOREM Spec ⇒ ∈TYPE_INVARIANTS

Interruttore

EXTENDS Naturals
VARIABLES stato,cond1,cond2

(* INIZIALIZZAZIONE *)

TYPE_INVARIANTS $\triangleq$ $\wedge$stato $\in$ {0,1}
$\wedge$cond1 $\in$ {0,1}
$\wedge$cond2 $\in$ {0,1}

INIT $\triangleq$ $\wedge$ stato=0
$\wedge$ cond1=0
$\wedge$ cond2=0

(* AZIONI *)

LOCAL luce $\triangleq$ $\wedge$ (cond1'= CASE (stato=0 $\wedge$ cond1=1 $\wedge$ cond2=0) $\rightarrow$ 0
TRUE $\rightarrow$ cond1)
$\wedge$ (cond2'= CASE (stato=0 $\wedge$ cond1=1 $\wedge$ cond2=0) $\rightarrow$ 0
TRUE $\rightarrow$ cond2)

LOCAL buio $\triangleq$ $\wedge$ (cond1'= CASE (stato=1 $\wedge$ cond1=0 $\wedge$ cond2=0) $\rightarrow$ 1
TRUE $\rightarrow$ cond1)
$\wedge$ (cond2'= CASE (stato=1 $\wedge$ cond1=0 $\wedge$ cond2=0) $\rightarrow$ 0
TRUE $\rightarrow$ cond2)

(* EVENTI *)

alza$\triangleq$ (stato'=CASE (stato=0 $\wedge$ cond1=1 $\wedge$ cond2=0) $\rightarrow$ 1 TRUE $\rightarrow$ stato)
$\wedge$ luce

abbassa $\triangleq$ (stato'=CASE (stato=1 $\wedge$ cond1=0 $\wedge$ cond2=0) -> 0 TRUE $\rightarrow$ stato)
$\wedge$ buio

Next $\triangleq$ $\vee$ alza $\vee$ abbassa

Spec $\triangleq$ INIT $\wedge$ $\lfloor$ Next$\rfloor_{<stato, cond1, cond2>}$

THEOREM Spec $\Rightarrow$ TYPE_INVARIANTS

main

```
EXTENDS  Naturals
VARIABLES  x,y,s1,s2

bambino ≜  INSTANCE Bambino WITH stato←s1, cond1←x, cond2←y
interruttore ≜ INSTANCE Interruttore WITH stato←s2, cond1←x, cond2←y
```

```
INIT ≜  ∧bambino!INIT
        ∧ interruttore!INIT
        ∧ x=0
        ∧ y=0
        ∧ s1=0
        ∧ s2=0

TYPE_INVARIANTS ≜        ∧ x ∈ {0,1}
                         ∧ y ∈ {0,1}
                         ∧ s1 ∈ {0,1}
                         ∧ s2 ∈ {0,1}
```

```
Next ≜  ∨ (bambino!Next ∧ interruttore!Next)

Spec≜  MInit ∧ ⌊ Next⌋<x,y,s1,s2>
```

Il modulo *main* è l'entità che consente l'effettiva interoperabilità tra i due sistemi: de facto esso non fa altro che istanziare due *oggetti,* uno di tipo *bambino* ed uno di tipo *interruttore* e, utilizzando le parti di questo, creare il proprio ciclo di vita; il suo stato iniziale prevede che gli stati iniziali dei due moduli siano in congiunzione, così come il predicato di safety Next, sì descritto per evitare comportamenti di deadlock di non applicazione di azioni.
È importante osservare come le condizioni siano condivise: è questa particolarità che consente l'evoluzione sincrona del sistema, abilitando prima un modulo e poi l'altro all'esecuzione di azioni.
Nell'azione TYPE_INVARIANTS le condizione vengono vincolate variare su un insieme binario, non permettendo ovviamente TLA+ di definire tipi di dato quali i bit, così come s1 ed s2, simboli di stato compiono un mapping perfetto sull'insieme degli stati.

TESTING AUTOMATICO

IL TOOL

TLC è un model checker utilizzato per vagliare specifiche ridotte in linguaggio TLA+; questo riceve in input un file che riporta i moduli TLA+ per compiere model checking su di esse.
I file che questo tool prende in input sono scritti in un linguaggio ASCII molto simile a TLA+, differisce in realtà la sola rappresentazione figurativa dei simboli, ad esempio il simbolo di definizione viene scritto come == e non come ≙ , oppure il simbolo di diverso è scritto come #.
Altro input di TLC è un file di configurazione, che non possiede alcuna collusione colla logica di Lamport, che in sostanza indica al TLC i nomi delle specifiche e delle proprietà da verificare.
Il modo più efficace di trovare errori in una specifica è quello di provare a verificare che questa soddisfi le proprietà desiderate.
Possiamo utilizzare TLC senza dover verificare alcuna proprietà, nel qual caso esso andrà a cercare solo due tipi di errore: errori di *Silliness*, cioè errori di semantica, oppure errori di *Deadlock*, dei quali il TLC cerca di dimostrarne l'assenza.
Le specifiche e le proprietà che TLC verifica sono mere formule temporali.
Nella nostra trattazione, abbiamo preso in esame gli esempi di casi di studio proposti precedentemente in traduzione utilizzando il tool per verificare la correttezza delle nostre specifiche e dimostrare alcune proprietà, relative alle stesse.
In questa sezione la nostra premura sarà quella di riportare la traduzione dei sistemi in esame in linguaggio ASCII di TLC, mostrando successivamente, tramite printscreen, i risultati dell'esecuzione del tool.

PRIMO ESEMPIO: IL CLIENTE DELLA BANCA

Il primo esempio trattato è relativo al sistema del cliente bancario, il cui diagramma degli stati e delle transizioni e la relativa traduzione in TLA+ sono riportati nel capitolo precedente.
Riportiamo di seguito il testo del file di input .tla e del relativo file di configurazione:

clienteBanca.tla

```
--------------------------- MODULE ClienteBanca ---------------------------
EXTENDS Naturals
VARIABLES stato
-------------------------------------------------------------------------------
BInit == /\ stato = 0
BTypeInv ==   /\ stato \in {0, 1, 2, 3}
-------------------------------------------------------------------------------
paga == stato' = CASE stato = 0 -> 1 [] stato = 1 -> 1 [] stato = 2 -> 3 [] stato = 3 ->
3 [] TRUE ->stato
ritardo == stato' = CASE stato = 0 -> 2 [] stato = 1 -> 2 [] stato = 3 -> 2 [] TRUE ->
stato
BNext == \/ paga \/ ritardo
-------------------------------------------------------------------------------
BSpec == BInit /\ [][BNext]_stato
Continuapaga == []<>(paga)=>(stato=0 \/ stato =1) (*un cliente che continua a pagare
rimane affidabile o nuovo*)
MaipiuAff == []<>ritardo => <>[](stato#1)   (*un cliente in ritardo anche solo una volta
nn sarà mai piu affidabile*)
Primoevento == []<>(stato=0 /\ (paga \/ ritardo)) => []<>(stato'=1 \/ stato'=2)(*dopo il
primo evento il cliente è affidabile o moroso*)
Fake == []<> (\lnot paga) => []<> (stato'=1) (*propietà falsa: se un cliente non paga
resta affidabile*)
Dim == Continuapaga (*MaipiuAff Primoevento Fake*)
-------------------------------------------------------------------------------
THEOREM BSpec => []BTypeInv
===============================================================================
```

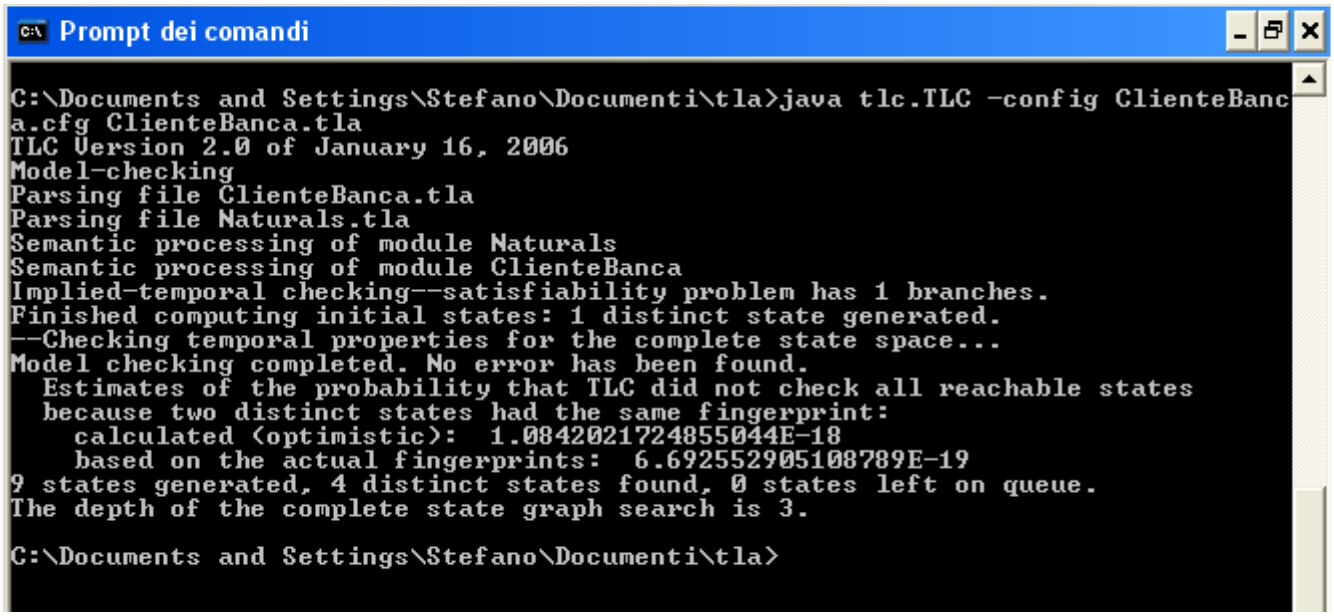
clientebanca.cfg

```
SPECIFICATION BSpec
INVARIANT BTypeInv
PROPERTY  Dim
```

Come possiamo vedere il file di configurazione ci dice che vogliamo dimostrare la proprietà Dim la quale, di volta in volta, sarà rappresentativa delle nostre intenzioni.

PRIMA PROPRIETÀ: `[]<>(STATO=0 /\ (PAGA \/ RITARDO)) => []<>(STATO'=1 \/ STATO'=2)`

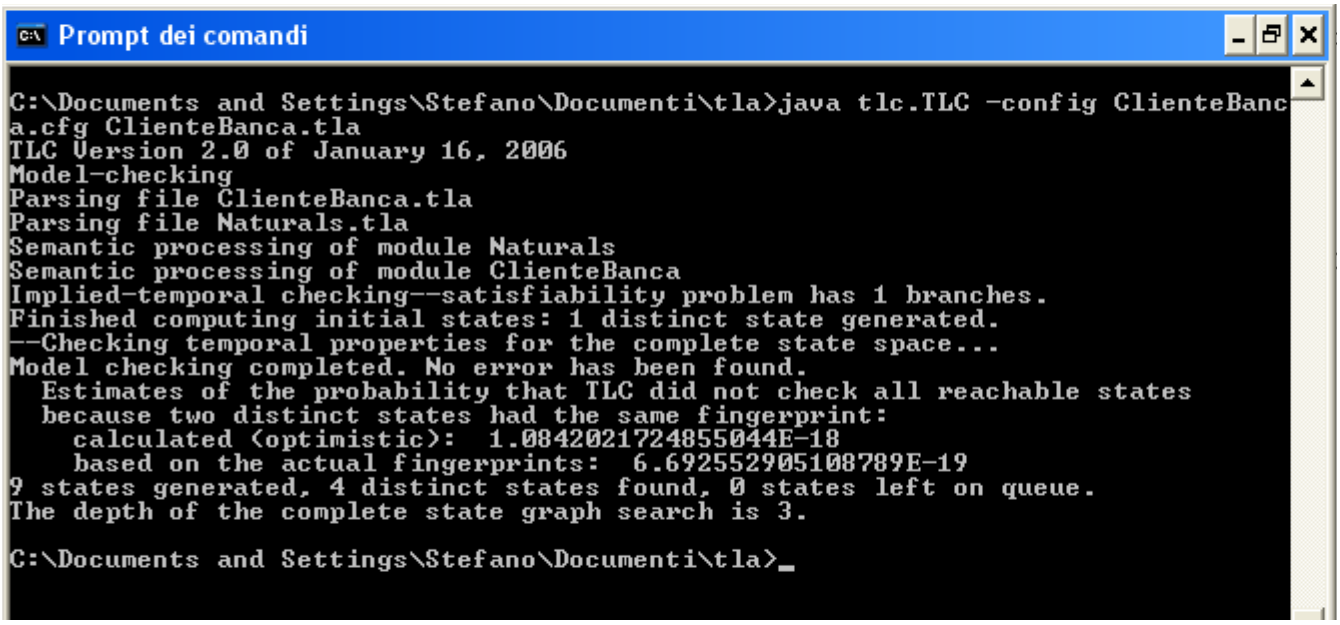
Questa proprietà afferma che dopo il primo evento il cliente è affidabile o moroso; vediamo l'esecuzione di TLC



Come possiamo vedere il tool che il model checking è completo e non sono stati trovati errori.

SECONDA PROPRIETÀ: `[]<>(PAGA)=>(STATO=0 \/ STATO =1)`

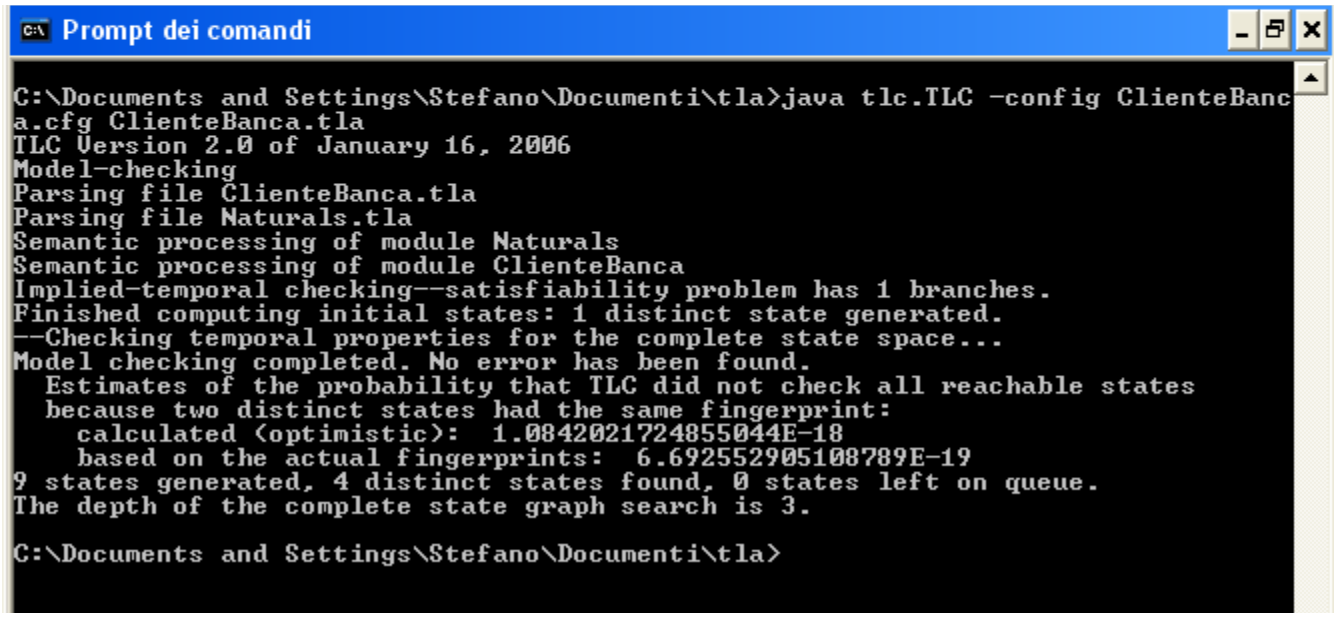
Questa proprietà afferma che un cliente che continua a pagare rimane affidabile o nuovo; vediamo l'esecuzione di TLC



Anche in questo caso l'esito del model checking è positivo

TERZA PROPRIETÀ: `[]<>RITARDO => <>[](STATO#1)`

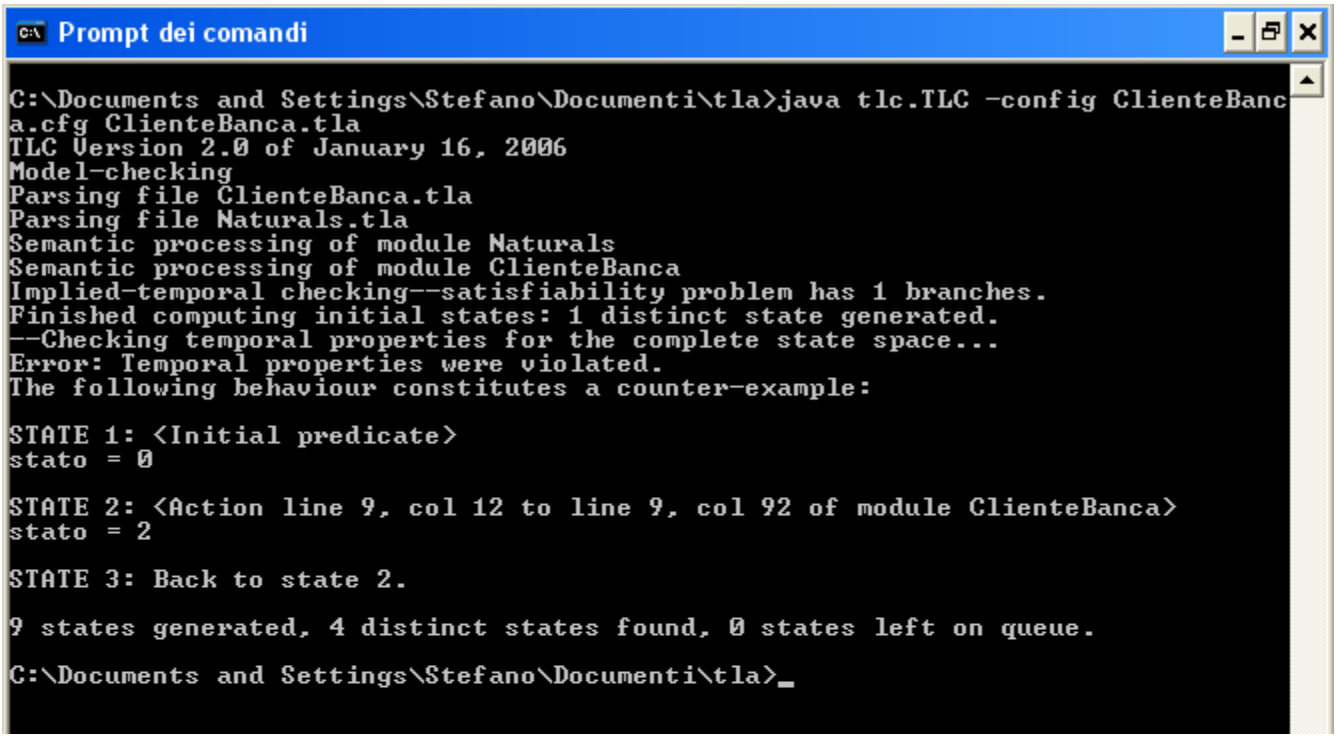
Questa proprietà afferma che un cliente in ritardo anche solo una volta non sarà mai più affidabile; vediamo TLC



Anche in questo caso il tool verifica correttamente la proprietà

QUARTA PROPRIETÀ: []<> (\LNOT PAGA) => []<> (STATO'=1)

Questa è una proprietà chiaramente falsa che abbiamo volutamente scritto per provare un caso in cui TLC dia risposta negativa; in sostanza afferma che se un cliente non paga resta affidabile; vediamo come risponde TLC



Come possiamo notare in questo caso il tool ci risponde che le proprietà temporali sono state violate.

SECONDO ESEMPIO: IL BAMBINO E L'INTERRUTTORE

Il secondo esempio che prendiamo in considerazione è quello del bambino e l'interruttore, anch'esso ampiamente descritto nel capitolo relativo alle traduzioni; riportiamo il codice del file tla e del relativo file di configurazione

bambino.tla

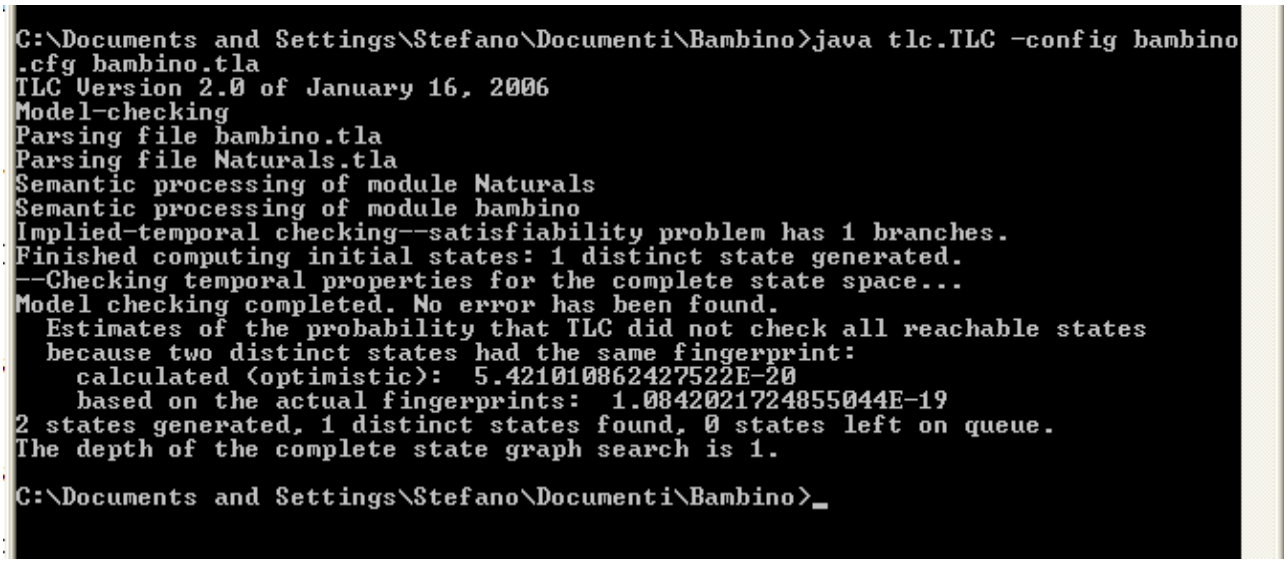
```
----------------------------------MODULE Bambino---------------------------------------
EXTENDS  Naturals
VARIABLES stato
CONSTANTS alza , abbassa
ASSUME /\ alza \in Nat
          /\ abbassa \in Nat
-------------------------------------------------------------------------------------
------
(* mapping degli stati:
       stato=0 -> contento
       stato=1 -> scontento
*)

Inv ==        stato \in {0,1}

BInit == stato=0

buio ==       (stato'= CASE stato=0 -> 1 [] TRUE -> stato)
              /\ alza' = CASE stato=0 ->1 [] TRUE ->alza

luce ==       (stato'= CASE stato=1 -> 0 [] TRUE -> stato)
              /\ abbassa'= CASE stato=1 ->0 [] TRUE ->abbassa

BNext == \/ buio \/ luce
LOCAL try == buio /\ luce
AlternaSpec == <>[]try => <>[](stato=0 <=> stato'=1)
-------------------------------------------------------------------------------------
BSpec == BInit /\ [][BNext]_stato
THEOREM BSpec => (*[]AlternaSpec*) []Inv
```

```
============================================================================================
```

bambino.cfg

```
SPECIFICATION BSpec
INVARIANT Inv
CONSTANTS alza=0
               abbassa=0
PROPERTY AlternaSpec
```

Come notiamo il file di configurazione dice al tool che è interessato a dimostrare la proprietà AlternaSpec, che afferma che il bambino alterna tra contento e scontento ad ogni istante; vediamo l'esecuzione di TLC



Vediamo chiaramente che il tool dà risposta positiva, affermando che la nostra formula temporale non contiene errori.

TERZO ESEMPIO: IL BAMBINO E L'INTERRUTTORE, VERSIONE SINCRONA

In questo caso ci proponiamo di studiare il sistema del bambino e l'interruttore in versione sincrona, andando a definire tre moduli: uno che rappresenta il bambino, uno che rappresenta l'interruttore ed infine un modulo main che si presenta come punto di raccordo tra i due moduli; questo de facto permette la creazione del sistema sincrono:

bambino.tla

```
----------------------------------MODULE Bambino-------------------------------------
EXTENDS  Naturals
VARIABLES cond1, cond2,stato
-------------------------------------------------------------------------------------
(* mapping degli stati:
       stato=0 -> contento
       stato=1 -> scontento
*)

BInv == stato \in {0,1}
             /\ cond1 \in {0,1}
             /\ cond2 \in {0,1}

BInit == stato=0 /\ cond1=0 /\ cond2=0
-------------------------------------------------------------------------------------
```

```
(* AZIONI *)

LOCAL alza == /\ (cond1'=CASE (stato=0 /\ cond1=0 /\ cond2=1) ->1 [] TRUE ->cond1)
                     /\ (cond2'=CASE (stato=0 /\ cond1=0 /\ cond2=1) ->0 [] TRUE ->
cond2)

LOCAL abbassa == /\(cond1'=cond1)
                          /\(cond2'=CASE (stato=1 /\ cond1=1 /\ cond2=1) ->1 [] TRUE
->cond2)
------------------------------------------------------------------------------------
(* EVENTI*)

buio ==      /\(stato'= CASE (stato=0 /\ cond1=0 /\ cond2=1)->1 [] TRUE -> stato)
             /\ alza


luce ==      /\(stato'= CASE (stato=1 /\ cond1=1 /\ cond2=1)-> 0 [] TRUE -> stato)
             /\ abbassa

BNext == \/ buio \/ luce
------------------------------------------------------------------------------------
BSpec == BInit /\ [][BNext]_{stato, cond1, cond2}
THEOREM BSpec => []BInv
====================================================================================
```

Interruttore .tla

```
---------------------------------MODULE Interruttore-----------------------------------
EXTENDS Naturals
VARIABLES stato,cond1,cond2

------------------------------------------------------------------------------------
----
(* INIZIALIZZAZIONE *)

Inv == /\stato \in {0,1}
          /\cond1 \in {0,1}
          /\cond2 \in {0,1}

IInit == /\ stato=0
              /\ cond1=0
              /\ cond2=0
------------------------------------------------------------------------------------
(* AZIONI *)

LOCAL luce == /\ (cond1'= CASE (stato=0 /\ cond1=1 /\ cond2=0) -> 0 [] TRUE -> cond1)
                     /\ (cond2'= CASE (stato=0 /\ cond1=1 /\ cond2=0) -> 0 [] TRUE ->
cond2)

LOCAL buio == /\ (cond1'= CASE (stato=1 /\ cond1=0 /\ cond2=0) -> 1 [] TRUE -> cond1)
                     /\ (cond2'= CASE (stato=1 /\ cond1=0 /\ cond2=0) -> 0 [] TRUE ->
cond2)
------------------------------------------------------------------------------------
(* EVENTI *)

alza == (stato'=CASE (stato=0 /\ cond1=1 /\ cond2=0) -> 1 [] TRUE -> stato)
             /\ luce

abbassa == (stato'=CASE (stato=1 /\ cond1=0 /\ cond2=0) -> 0 [] TRUE -> stato)
                   /\ buio

INext == \/ alza \/ abbassa
------------------------------------------------------------------------------------
ISpec == IInit /\ [][INext]_{stato, cond1, cond2}

THEOREM ISpec => []Inv

====================================================================================
```

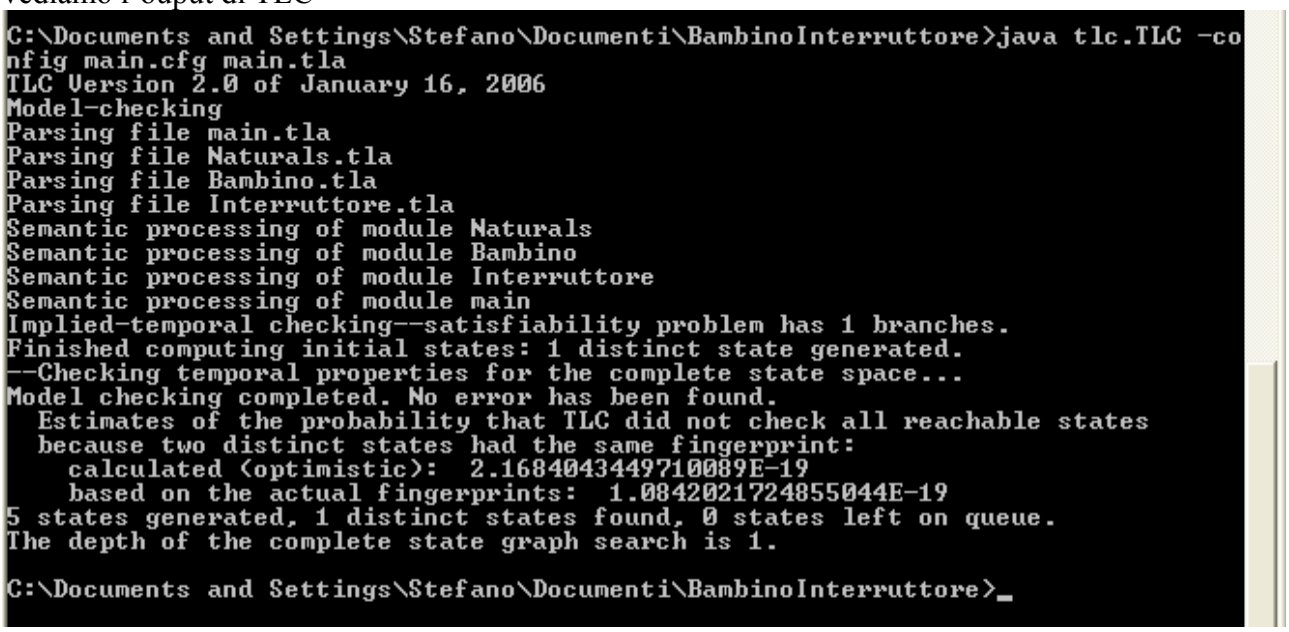
main .tla

```
----------------------------MODULE main-----------------------------------------------------
EXTENDS Naturals
VARIABLES x,y,s1,s2

bambino == INSTANCE Bambino WITH stato<-s1, cond1<-x, cond2<-y
interruttore == INSTANCE Interruttore WITH stato<-s2, cond1<-x, cond2<-y

MInit == bambino!BInit /\ interruttore!IInit /\ x=0 /\ y=0 /\ s1=0 /\ s2=0

MInv == /\ x \in {0,1}
             /\ y \in {0,1}
             /\ s1 \in {0,1}
             /\ s2 \in {0,1}

-------------------------------------------------------------------------------------------

Next ==  \/ bambino!BNext /\ interruttore!INext

Spec== MInit /\ [][Next]_{x,y,s1,s2}

try == <>[] \neg interruttore!abbassa => <>[](x'=0 /\ y'=0)
===========================================================================================
```

main.cfg

```
INIT MInit
NEXT Next
INVARIANT MInv
PROPERTY try
```

La proprietà che vogliamo verificare, come riportato nel file di configurazione, abbiamo deciso di chiamarla try e in sostanza rappresenta la stessa proprietà dimostrata per l'esempio precedente; vediamo l'ouput di TLC



Come si vede il tool dà un risultato positivo anche in questo caso.

TRADUZIONE DEI PROGRAMMI

INTRODUZIONE

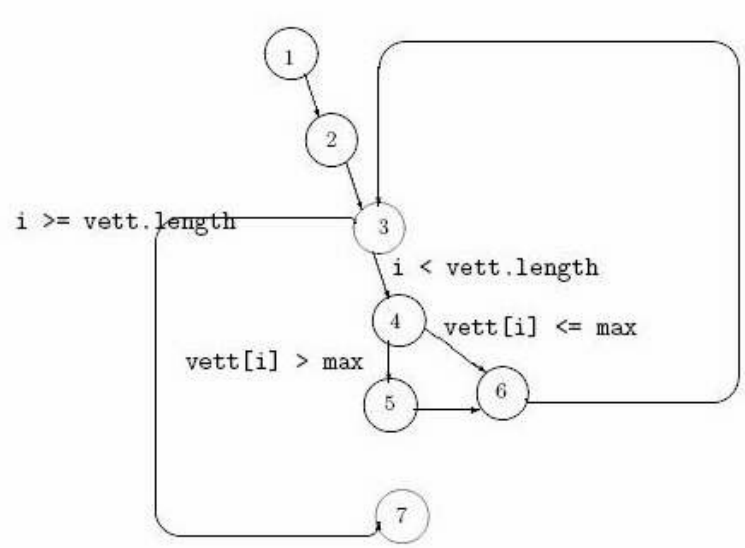
In questo capitolo decidiamo di occuparci della traduzione degli algoritmi, redatti nella logica temporale di Lamport; il significato di questo paragrafo è quello di spiegare la suddivisione dei paragrafi che segue: l'uso proposto dall'autore del suo formalismo è quello di compiere testing sulle specifiche e non sulla realizzazione delle stesse, mentre per noi questo è il fine.
Per completezza presenteremo entrambe le nostre metodologie, quella di Lamport, in forma breve e solo accennata rimandando, a chi è interessato, ai testi più prolissi dell'autore, e la nostra che si occupa della traduzione delle realizzazioni alla stregua di una macchina a stati.

TRADUZIONE DI ALGORITMI IN TLA+

Sì da essere completi nella nostra traduzione proporremo la nostra metodologia di traduzione per mezzo di un esempio, esempio di programma è quello banale di calcolare il valore massimo degli elementi di un array.

```
public static int Massimo(int [] vett)
{
     int result= MIN;                    /*1*/
     int i=0;                            /*2*/
     while(i<vett.length)                /*3*/
     {
          if(vett[i]>result)             /*4*/
               result=vett[i];           /*5*/
          i++;                           /*6*/
     }
     return result;                      /*7*/
}
```

Tracciamo il grafo di controllo dell'algoritmo scritto e riportiamolo di seguito:



Per motivi di semplicità rappresentativa ipotizziamo che gli algoritmi dei quali ci occupiamo abbiano come forma quella di Massimo,cioè i nodi iniziali del grafo, qui 1 e 2, sono stati di inzializzazione delle variabili d'uso; tale semplificazione, non necessaria ai fini generali ma per

motivi esclusivamente di brevitas, ci permette di raggruppare gli stati 1 e 2 nel predicato INIT di un modulo TLA+.
Prima di essere così precipitosi è bene dire che il nostro algoritmo verrà integrato in un modulo TLA+ possidente medesimo nome, il quale de facto sarà utilizzato come una simulazione del grafo di controllo.
Variabili del modulo saranno lo *stato,* l'indice *i,* utilizzato nella scansione del ciclo, e *result*, variabile d'appoggio che indica il valore di ritorno; è importante da aggiungere che i grafi dei quali ci occupiamo possiedono uno ed un solo punto di uscita, cioè un solo nodo può essere considerato come lo stato finale del nostro automa.
Costanti saranno utilizzate per rappresentare: il vettore in ingresso *vett*, la sua lunghezza, che chiameremo N, ed il valore minimo di rappresentazione degli interi MIN , difatti TLA+ assume gli interi come illimitati superiormente ed inferiormente, questo perché non c'è possibile compiere testing su istanze generiche. Il ricorso alle costanti permette di parametrizzare ovviamente il comportamento del sistema modellato.
Etichettiamo ogni arco del grafo di controllo con una stringa, la quale sarà utilizzata come identificativo di un'azione TLA+, azione che possiederà restrizioni di visibilità LOCAL, sì da rispettare il principio del minimo privilegio; le azioni che ci apprestiamo a descrivere non faranno altro che rappresentare l'evoluzione del grafo nelle sue variabili *stato*, *i* e *result,* perciò riconosceremo i salti condizionali delle istruzioni if e while, coì come le assegnazioni e le somme.
Mostriamo di seguito quanto da noi realizzato:

---**Massimo**---

```
EXTENDS Naturals, Sequences

VARIABLES i, stato, result

CONSTANT MIN,N

----------------------------------------------------------------------------

(* INIZIALIZZAZIONE *)

 vett ≜ <<3,15,6>>

Minv≜        ∧ i ∈ Nat
             ∧ stato ∈ {1,2,3,4,5,6,7}
             ∧ N ∈ Nat
             ∧ N>0

MInit≜       ∧ result=MIN
             ∧ i=1
             ∧ stato=3

----------------------------------------------------------------------------

(* TRANSIZONI DEL GRAFO *)

LOCAL testWhile ≜  ∧ (stato'= CASE (stato=3 ∧ (i = <N)) → 4
                        (stato=3 ∧ (i > N)) → 7
                        TRUE → stato)
                   ∧ result'=result
                   ∧ i'=i

LOCAL testIf≜  ∧ (stato'= (CASE (stato=4 ∧ (vett[i]> result))→5
```

(stato=4 ∧ (vett[i] \leq result))→6
TRUE → stato))
∧ result'=result
∧ i'=i

LOCAL assegnazione≙ ∧ (stato'=(CASE stato=5 →6
TRUE→ stato))
∧ (result'= (CASE stato=5→ vett[i]
TRUE→result))
∧ i'=i

LOCAL incI≙ ∧ (stato'= CASE (stato=6)→3
TRUE → stato)
∧ (i'=CASE (stato=6)→ i+1
TRUE →i)
∧ result'=result

LOCAL return ≙ ∧ stato'= stato
∧ result'=result
∧ i'=i

Next ≙ testWhile ∨ testIf ∨ assegnazione ∨ incI ∨ return

Mspec≙ MInit ∧ $\lfloor$ Next$\rfloor_{<result,i,stato>}$ ∧ (i ≤ N)

Osserviamo come la traduzione dell'algoritmo in questo caso preveda anche, nella specifica, l'introduzione di condizioni di liveness, ovvero di comportamenti che devono accadere per far evolvere il sistema; in questo caso per evitare comportamenti anomali obblighiamo l'indice i di scansione del vettore a mai superare la lunghezza del vettore stesso, altrimenti si ignora cosa leggerebbe; i questo contesto è bene aggiungere che per TLA+ i vettori vengono letti dalla cella 1 alla cella N, in pratica il primo elemento è l'elemento 1, non lo 0 come nei normali linguaggi di programmazione, è questo il motivo della particolare scrittura dell'azione *testWhile.*

TESTING AUTOMATICO

Il testing automatico che ci proponiamo di compiere è sull'algoritmo Massimo nella specifica in TLA+

Riportiamo di seguito i file di input tla e il file di configurazione.

Massimo.tla

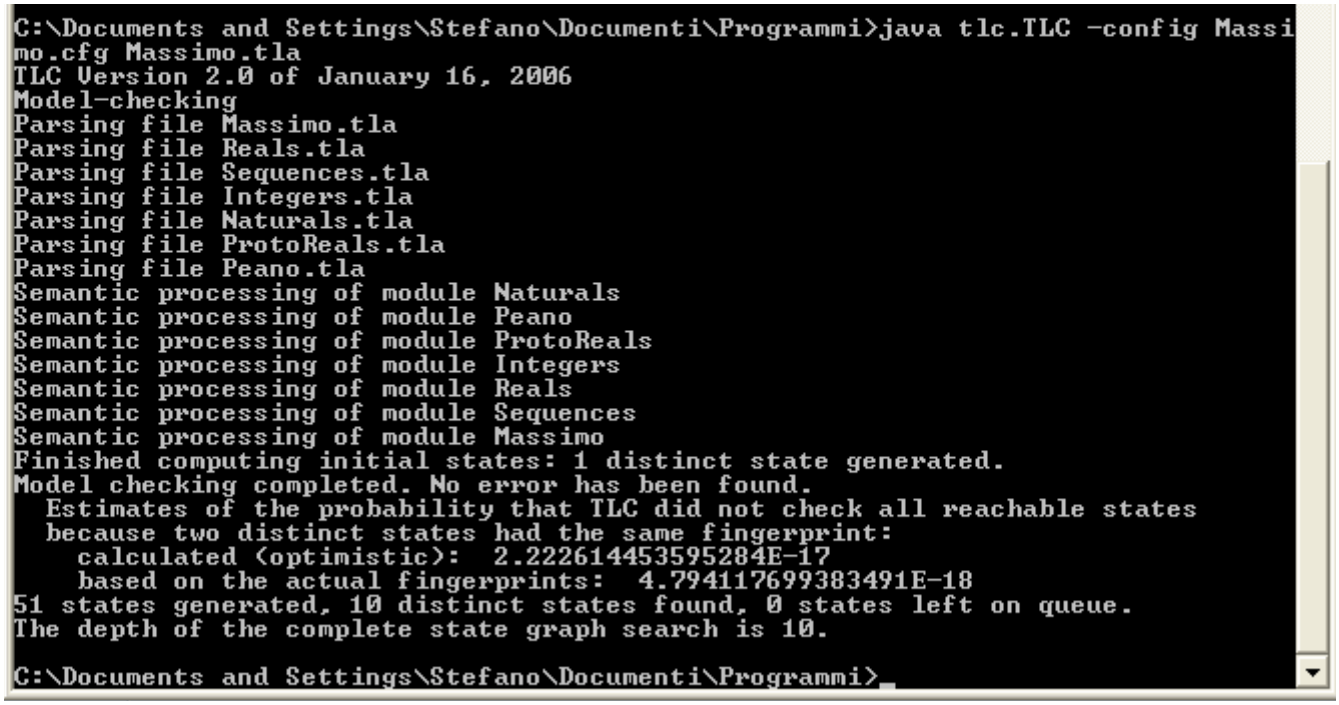
```
----------------------------MODULE Massimo-------------------------
EXTENDS Naturals, Sequences

VARIABLES i, stato, result(*, vett*)

CONSTANT MIN,N
-------------------------------------------------------------------------------
-------------
(* INIZIALIZZAZIONE *)

 vett == <<3,6,15>>

MInv == /\ i \in Nat
        /\ stato \in {1,2,3,4,5,6,7}
         /\ N \in Nat /\ N>0

MInit == /\ result=MIN
        /\ i=1
        /\ stato=3

-------------------------------------------------------------------------------
(* TRANSIZONI DEL GRAFO *)

LOCAL testWhile == /\ (stato'= CASE (stato=3 /\ (i \leq N))->4 []
                       (stato=3 /\ (i > N))->7[]
                        TRUE->stato)
                   /\ result'=result
                   /\ i'=i

LOCAL testIf == /\(stato'= (CASE (stato=4 /\ (vett[i]> result))->5 []
                  (stato=4 /\ (vett[i] \leq result))->6 []
                  TRUE -> stato))
                /\ result'=result
                /\ i'=i

LOCAL assegnazione == /\ (stato'=(CASE stato=5 ->6 [] TRUE-> stato))
                      /\ (result'= (CASE stato=5-> vett[i][] TRUE->result))
                      /\ i'=i

LOCAL incI == /\ (stato'= CASE (stato=6)->3 [] TRUE -> stato)
              /\ (i'=CASE (stato=6) ->i+1 [] TRUE ->i)
              /\ result'=result

LOCAL return == /\ (stato'= CASE (stato=7 \/ stato=8)->8 [] TRUE -> stato)
                /\ result'=result
                /\ i'=i

Next == testWhile \/ testIf \/assegnazione \/incI \/return
-------------------------------------------------------------------------------
------------
```

```
Mspec == MInit /\ [][Next]_{stato,i,result}

Terminata == <>[](stato=8)
Pre == \A x \in {1,2,3}: vett[x]>0
Occorrenza == \E x \in {1,2,3}: (vett[x]>0 /\ vett[x] = result)
Maggiore == \A x \in {1,2,3}: result >= vett[x]
Post == Occorrenza /\ Maggiore
Term == (Pre => <> Terminata ) (* terminazione*)
CParz == (Pre => [](Terminata =>Post)) (* correttezza parziale*)
CTot == (Pre => ((<> Terminata) /\ ([]Terminata =>[]Post))) (*corr. Totale*)
Prova == (Terminata =>Post)
=============================================================================
```

Massimo.cfg

```
SPECIFICATION Mspec
CONSTANT  MIN=0
          N =3
PROPERTY Term
PROPERTY CParz
PROPERTY CTot
```

Vogliamo verificare le proprietà di terminazione, correttezza parziale e correttezza totale: come si può notare abbiamo riportato queste proprietà nel file di configurazione; presentiamo l'output di TLC nei tre casi:



correttezza parziale e totale.

Si riscontrano in realtà problemi colla verifica automatica della terminazione; il tool de facto torna un comportamento, behaviour, quale controesempio che in realtà dimostra la nostra specifica.

PROBLEMI RISCONTRATI

Nell'esecuzione del tool relativa alla specifica del programma java che trovava il massimo all'interno di un array, abbiamo riscontrato il problema di non riuscire a bloccare il non determinismo che caratterizza TLC. In sostanza, il tool seguiva in modo non deterministico una serie di stati rappresentati tramite un grafo, non riuscendo a dimostrare le proprietà di correttezza e terminazione