

Applicazione industriale di metodi formali:
attuali tendenze
nelle normative internazionali
e negli strumenti di supporto.

A. Fantechi
Dipartimento di Sistemi e Informatica
Università di Firenze

Metodi formali: alcune definizioni

tecniche rigorose e basate su formalismi matematici per la specifica e la verifica del software

[D. Craigen]

”the application of mathematical synthesis and analysis techniques to the development of computer controlled systems”.

“a software specification and production method that comprises:

- a collection of mathematical notations addressing the specification, design and development phases;
- a well-founded logical inference system in which formal verification proofs and other properties can be formulated;
- a methodological framework within which software can be developed from the specification to the implementation in a formally verifiable manner”

Scopo dell'utilizzo dei M.F.: “Correttezza” del prodotto software

Semantica dei linguaggi di programmazione

Prime applicazioni dei metodi formali:

Dare la *semantica formale di linguaggi di programmazione*

Una volta data la semantica formale, si può in teoria dimostrare che un programma è corretto, cioè esegue la funzione che la sua specifica (formale) richiedeva.

- > Infattibile in pratica per programmi non semplici
 - > Prove non facilmente supportabili da strumenti
 - > La semantica formale di un linguaggio di programmazione è spesso scarsamente utilizzabile per la sua mole
-
- Utilizzo di VDM per il linguaggio Ada (anni '80)
 - Utilizzo di ASM per Java, SDL (2000)

Applicazioni industriali

- Problemi di accettazione dovuti a:
 - complessità notazionale
 - scarsità di strumenti di supporto
 - difficoltà di scelta di un metodo tra i tanti proposti

- Utilizzo praticamente limitato ai settori:
 - safety-critical
(ferroviario, avionico/spaziale, automotive, medicale, nucleare,....)
 - telecomunicazioni
(l'interoperabilità tra prodotti di diversa provenienza può essere ottenuta solo con la stretta aderenza a standard non ambigui)

Metodi *asserzionali* per la specifica e lo sviluppo formale

- un sistema e' visto come un insieme di stati, e di operazioni che modificano lo stato
- Viene definito un INVARIANTE, un predicato che deve essere soddisfatto in tutti gli stati
- Per ogni operazione viene definito un predicato detto PRECONDIZIONE. L'operazione puo' essere effettuata solo se la sua preconditione e vera sullo stato corrente.
- Per ogni operazione viene definito un predicato detto POSTCONDIZIONE. Questo predicato e' vero sullo stato modificato dall'operazione.

VDM, Z, B

```
i=0;
while (i<n)
{
  A[i] = 0;
  i++;
}
```

$i == 0$ preconditione

$0 < i \leq k \Rightarrow A[i-1] == 0$ Invariante (al k-esimo ciclo)

$0 < i \leq n \Rightarrow A[i-1] == 0$ postcondizione

- Un *invariante* viene dimostrato per induzione:
- Si dimostra che la preconditione implica l'invariante al primo passo;
- Si dimostra che se l'invariante vale al passo k , allora vale anche al passo $k+1$.
- Si dimostra poi che l'invariante al passo finale implica la postcondizione.
- *Le dimostrazioni avvengono sulla base della conoscenza della semantica di comandi del linguaggio di programmazione.*

A success story: the B method

*(based on Abstract Machines -
Jean-Raymond Abrial, BP Research, Oxford Programming Research Group)*

-ASSERTIONAL METHOD

- to simplify the development of the system the method uses the same notation for all the stages of the development, that goes on for successive refinement steps
 - verification amounts to check that the preconditions and postconditions of lower level operations verify the preconditions and postconditions of higher level operations

Basic concepts:

Set theory, predicate calculus.

- Addresses in particular the specification of sequential systems
- Data are easily formalized

An **Abstract Machine (AM)** is given by

- STATE
 - variable set
 - *STATE INVARIANT*, to be permanently verified
- a set of OPERATIONS that can be activated to modify the STATE
 - defined in terms of pre-conditions and post-conditions on the STATE

PROOF OBLIGATIONS:

every time that an operation has been specified, it must be verified that its specification preserves the *STATE INVARIANT*

The method includes :

- logic system for the expression and the proof of PROOF OBLIGATIONS based on the SUBSTITUTION principle

OPERATIONS:

modify the STATE within the constraints imposed by the INVARIANT

PROOF OBLIGATION

→ it must be proved that the specification of the operation preserves the invariant

ASSUMPTION:

the STATE verifies the INVARIANT before the operation

PROOF:

the INVARIANT is satisfied after the operation

A **REFINEMENT** machine is defined by adding implementation details to a machine

the preconditions and postconditions of *lower level operations* must verify the preconditions and postconditions of *higher level operations*

→ A **PROOF OBLIGATION** is generated to carry on this verification.

- The development methodology is made up of a sequence of refinement steps: at each step, the **PROOF OBLIGATIONS** have to be proved (with the support of a theorem prover)
- At a certain step of refinement, the machine is very close to an implementation, so it can be translated (almost automatically) into code.

Applicazioni ferroviarie di B

B è stato applicato con successo a diverse applicazioni di segnalamento ferroviario, in ambito MATRA Transport e ALSTOM, principalmente in Francia.

La prima applicazione si è avuta alla fine degli anni ottanta al SACEM, sistema di controllo della RER A parigina. B è stato introdotto a progetto già avviato per poterne portare a termine la validazione, praticamente su richiesta dei committenti (SNCF e RATP)

Sulla base di tale prima applicazione il metodo si è sviluppato, con la realizzazione di strumenti di supporto e con la esperienza guadagnata in altre applicazioni simili.

L'ultima importante applicazione del B si è avuta per la metropolitana METEOR, integralmente automatica, linea 14 di Parigi

L'ultima importante applicazione del B si è avuta per la metropolitana METEOR, integralmente automatica, linea 14 di Parigi



Meteor

- The new metro line, number 14, in Paris (France), is in operation since October 15th 1998 between Tolbiac-Nationale at south-east of Paris and Madeleine at north-west.
- This line was designed to reach traffic of 40.000 passengers per hour with an interval between train down to 85 s. on peak hours.
- This new line is managed by the automatic train operation system developed by Matra Transport International.

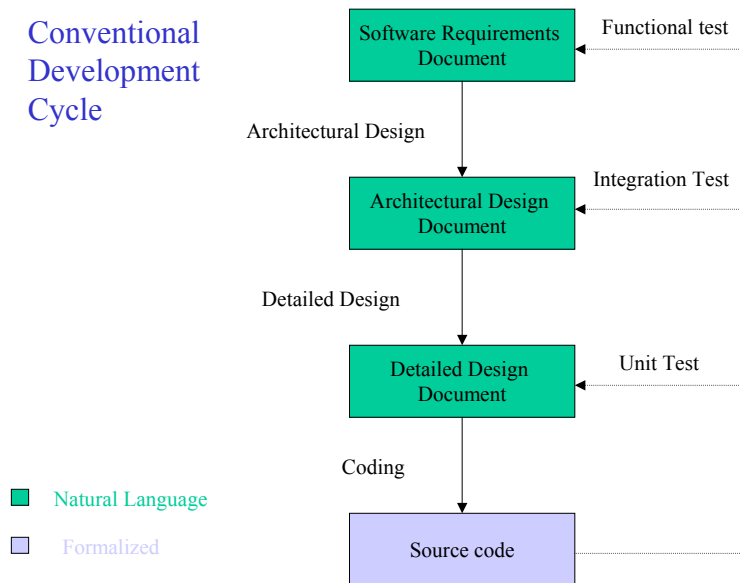
<i>Software Product</i>	<i>Lines of Ada</i>	<i>Instructions of Ada</i>	<i>Ada Packages</i>
Wayside	37,000	22,000	305
On-Board	30,000	20,000	160
Line	19,000	15,000	165
<i>Total</i>	<i>86,000</i>	<i>57,000</i>	<i>630</i>

Errors Found by Proof. Many errors have been found during proof activities.

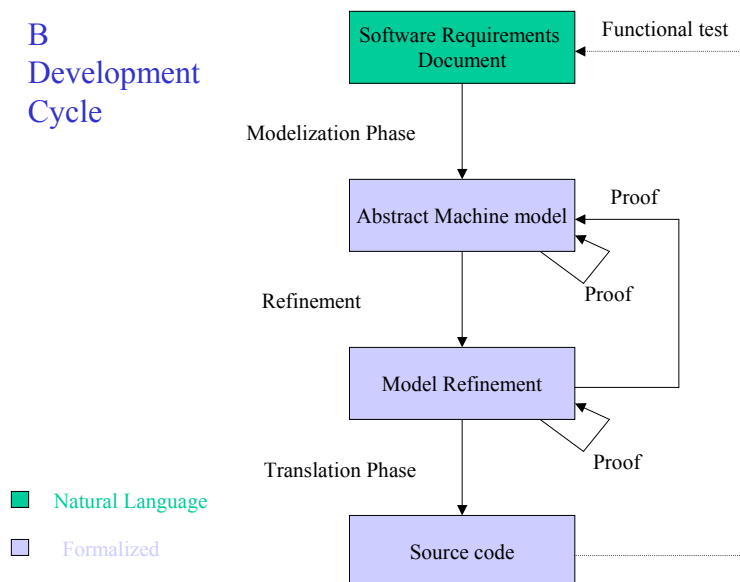
Errors Found by Testing.

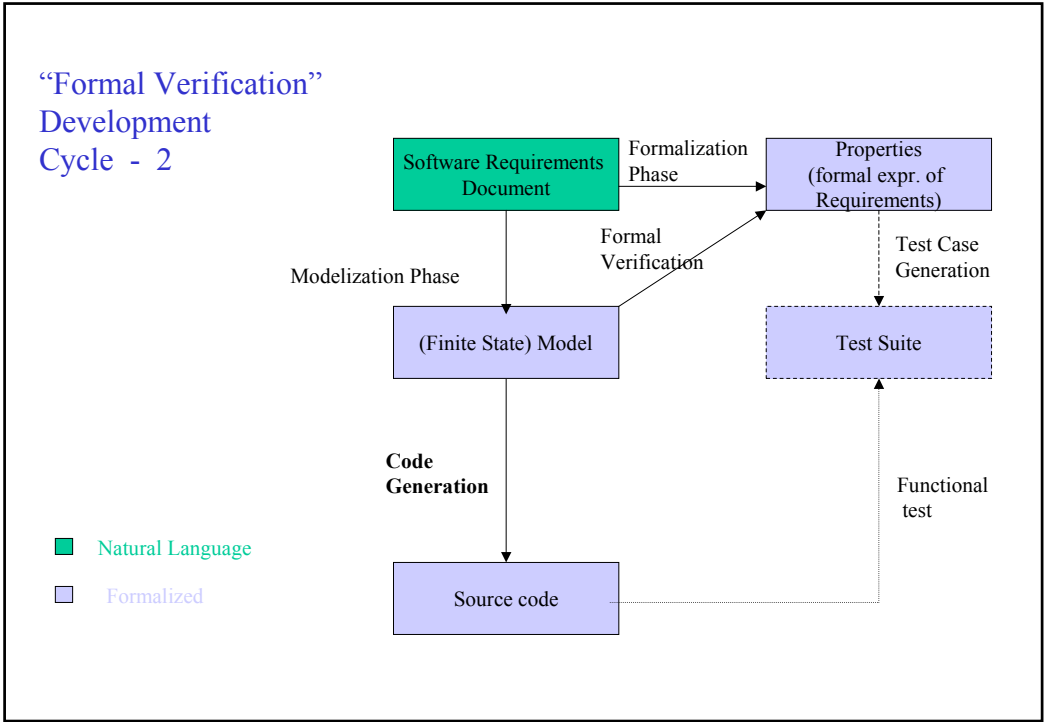
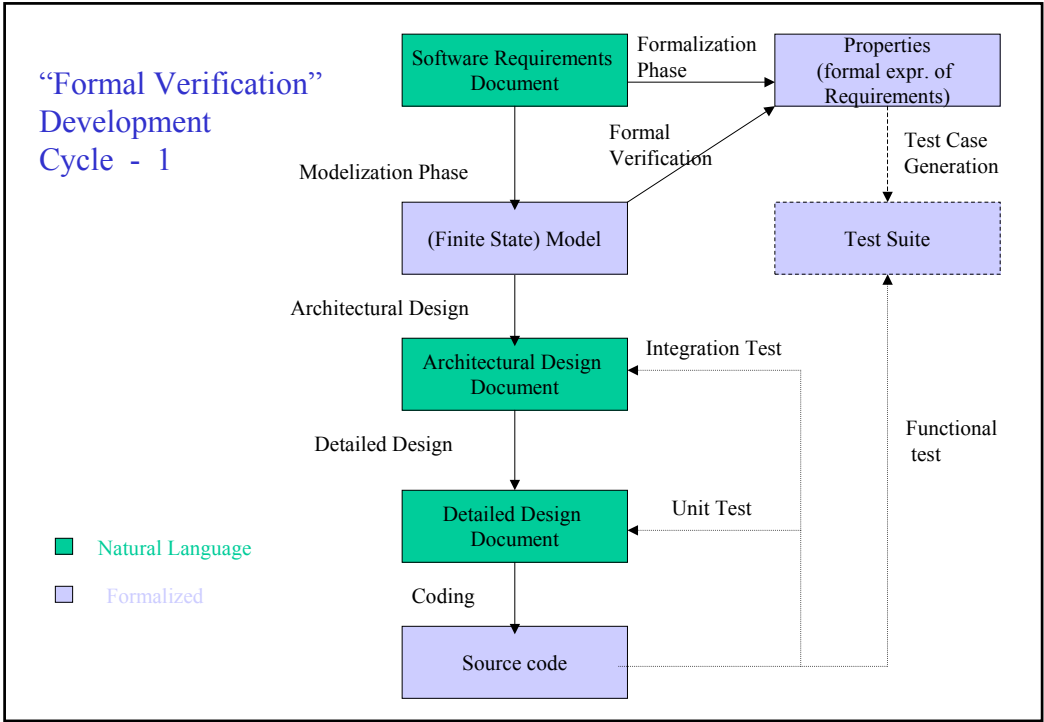
1. functional validation on host computer: no bug found;
2. integration validation on target computer: no bug found;
3. on-site tests: no bug found;
4. since the line operates: no bug found.

Conventional Development Cycle



B Development Cycle





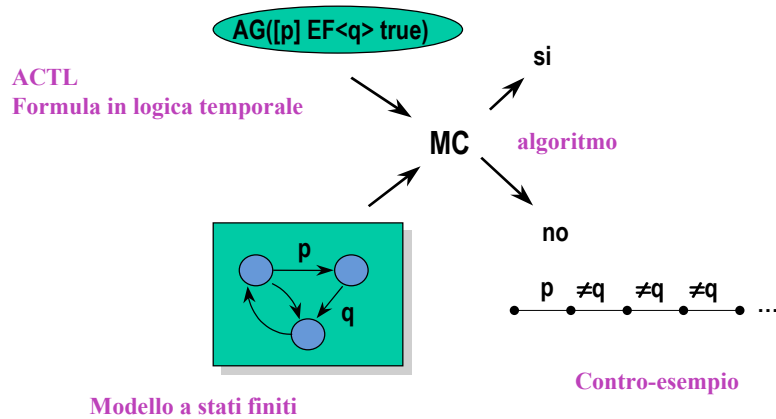
Verifica formale

- Sistema visto principalmente come macchina a stati
- (L'interesse è primariamente sul comportamento del sistema, piuttosto che sulla funzione o sui dati)
- La specifica è la specifica del comportamento del sistema
- Verifica se tale specifica soddisfa particolari requisiti (verifica di correttezza della specifica)
- Generazione del codice da specifica formale
- Generazione di casi di test dalla specifica formale
- (non ci poniamo nell'ottica di verificare i requisiti direttamente sul codice)

Theorem proving

- Prova di una proprietà su un modello mediante dimostrazione di teoremi
- Supportata da potenti strumenti di prova (es. PVS – SRI International).
- Richiede una notevole perizia perché lo strumento deve essere guidato
- Adatto quindi per esperti, meno adatto da essere utilizzato direttamente in un ambito industriale dove non tutti abbiano le necessarie conoscenze matematiche.

Model Checking *(Clarke/Emerson, Queille/Sifakis)*



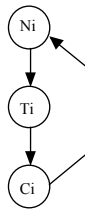
Il modello deve rappresentare **tutti** i comportamenti

L' algoritmo di model checking

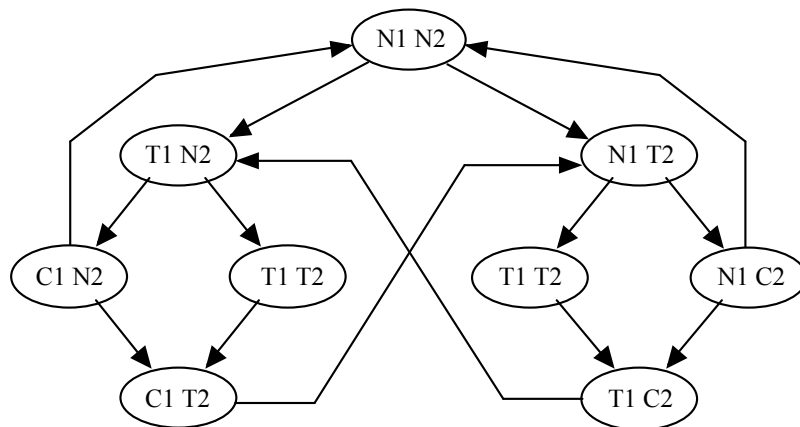
```

for i = 1 to length(p0)
for each subformula p of p0 of length i
case on the form of p
p = P, an atomic proposition /* nothing to do */
p = q and r: for each s in S
if q in L(s) and r in L(s) then add q and r to L(s)
end
p = ~q: for each s in S
if q in L(s) then add ~q to L(s)
end
p = EXq: for each s in S
if (for some successor t of s, q in L(t)) then add EXq to L(s)
end
p = A[q U r]: for each s in S
if r in L(s)
then add A[q U r] to L(s)
end
for j = 1 to card(S)
for each in S
if in L(s) and (for each successor t of s, A[q U r] in L(t))
then add A[q U r] to L(s)
end
end
p = E[q U r]: for each s in S
if r in L(s)
then add E[q U r] to L(s)
end
for j = 1 to card(S)
for each s in S
if q in L(s) and (for some successor t of s, E[q U r] in L(t))
then add E[q U r] to L(s)
end
end
end of case
end
end
    
```

Per studiare il funzionamento dell'algorithmo consideriamo il seguente esempio:
 due processi concorrono all'utilizzo di una risorsa comune; i due processi eseguono alcune operazioni in modo non critico (fase Ni, $i=1,2$), poi cercano di accedere alla risorsa (fase di "trying", Ti), e infine vi accedono, eseguendo alcune operazioni in modo critico (fase Ci), come schematizzato dal seguente diagramma degli stati.



Il problema è quello di implementare un meccanismo di mutua esclusione, che impedisca cioè il contemporaneo accesso dei due processi alla risorsa.

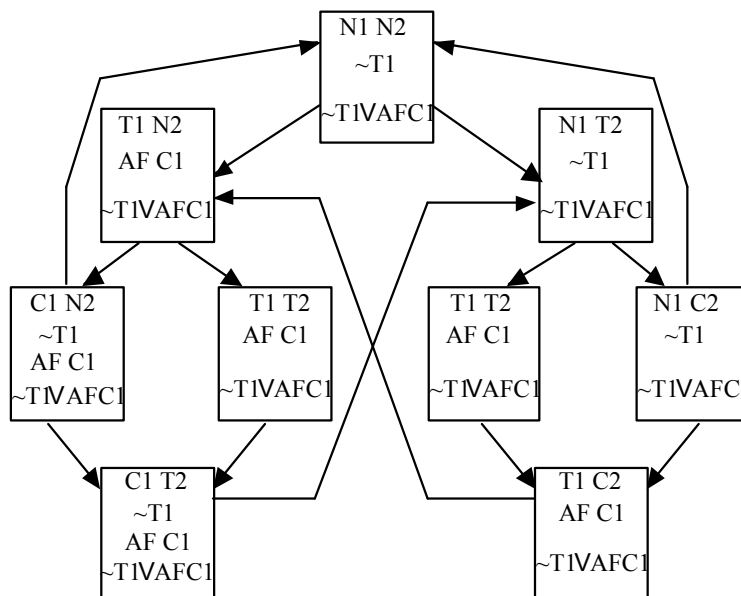


Si vede facilmente che questo meccanismo garantisce la mutua esclusione, perché non esiste alcuno stato in cui i due processi si trovano contemporaneamente nella fase critica - cioè verifica la formula CTL: $AG \sim(C1 \text{ and } C2)$

Ci si vuole assicurare però che il meccanismo non introduca "starvation", cioè se un processo entra nella fase Trying, prima o poi riesce a entrare nella fase critica, accedendo alla risorsa.

Si vuole cioè verificare la formula CTL: $AG(Ti \Rightarrow AF Ci)$,
cioè $AG(\sim Ti \text{ or } AF Ci)$

L'algoritmo di etichettamento, che parte dal diagramma degli stati così come lo abbiamo dato in precedenza, cioè dove appaiono i predicati atomici (Ni, Ti, Ci), dapprima etichetta gli stati in cui valgono le formule di lunghezza 1 ($\sim Ti, AF Ci$), poi quelle di lunghezza 2 ($\sim Ti \text{ or } AF Ci$), producendo il seguente grafo:



- la complessità computazionale dell' algoritmo risulta essere nel caso peggiore (quello dell'Until):
 $O(\text{length}(p0) * \text{card}(S) * (\text{card}(S) + \text{card}(R)))$, dove $\text{card}(S)$ è il numero degli stati e $\text{card}(R)$ è il numero delle transizioni. L' algoritmo descritto in [3] migliora questa complessità adottando una strategia depth first, con l'ausilio di una pila, in modo da analizzare lo spazio degli stati in una singola passata: il tempo di esecuzione è quindi lineare in termini della grandezza della formula (usualmente abbastanza piccola) e lineare in termini dello spazio degli stati.
- l' algoritmo di etichettamento permette, in caso di risultato negativo, di trovare il "perchè" del risultato negativo, visto che è possibile rintracciare quegli stati, che, non verificando sottoformule significative, contribuiscono al fallimento della verifica; in generale, gli algoritmi di model-checking possono fornire un "controesempio", evidenziando ad esempio un cammino nel modello che non verifica la (sotto)formula.
- **Esplosione dello spazio degli stati:**
 - può essere esponenziale con il numero dei processi coinvolti, (prodotto degli spazi degli stati di ogni processo).
 - Limita fortemente la capacità degli algoritmi di model-checking a lavorare su esempi di sistemi "reali". In pratica, un algoritmo di model checking classico come quello qui descritto può arrivare a verificare non più di un milione di stati.

Rappresentazione implicita dello spazio degli stati.

In questa tecnica, lo spazio degli stati (stati e transizioni) viene rappresentato come una funzione di transizione che viene codificata come funzione binaria (dati due stati, il fatto che esista una transizione tra di loro dà vero, altrimenti dà falso). Questa funzione binaria può essere efficientemente codificata in strutture compatte chiamate Binary Decision Diagrams (BDD)

Anche le formule possono essere espresse tramite BDD, e il Model-Checking diventa quindi un operatore che combina due BDD in modo da ottenere un risultato booleano.

Con questa tecnica si è arrivati a trattare sistemi da 10^{E+23} a 10^{E+120} stati.

Altre tecniche che sono state studiate e applicate in vari Model Checkers per ridurre il fenomeno dell'esplosione degli stati sono:

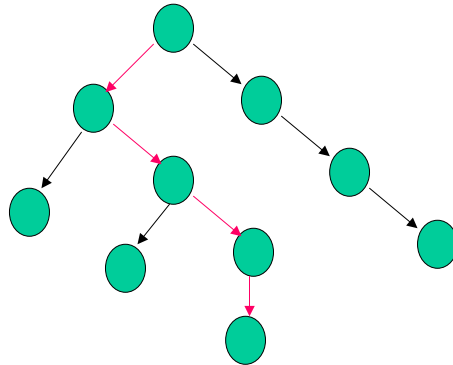
- Uso della simmetria del modello
- Tecniche di astrazione
- Ragionamento compositivo (Assume/Guarantee)
- Interpretazione astratta
- Minimizzazione del modello per equivalenza
- Riduzione per ordinamento parziale (Partial Order Reduction)
- Model checking "On the Fly" (lo stato globale dell'automa non viene costruito completamente prima di applicare l'algoritmo di etichettamento, ma vengono via via generate solo le regioni dell'automa che sono strettamente necessarie a verificare la formula).

La disponibilità di strumenti di Model Checking efficienti e capaci di lavorare su spazi degli stati di grandi dimensioni ne ha favorito, negli ultimi 6-7 anni, la diffusione anche in ambito industriale; nella seguente tabella troviamo elencati alcuni di quelli che vengono considerati come i maggiori successi di utilizzo del model checking; possiamo notare che in preponderanza si tratta di casi di verifica di dispositivi hardware e di protocolli.

1992	CMU	SMV	IEEE Futurebus Standard (1st time errors found in IEEE std.)
1992	Stanford	Murphi	Cache Coherence Protocol of IEEE Scalable Coherent Interface
1995	Bull & Verimag	CAESAR/ Aldebaran	LOTOS description of memory controller bus arbiter of PowerScale (PowerPC) Multiprocessor Architecture
1995		CWB	Anti-seismic active structural control (bug timing error)
1995	CMU	SMV	Analysis of Pentium FPU bug
1995 1996	Philips	Hytech Kronos	Consumer video recorder bus protocol
1996	Bell Labs/ AT&T	FormalCheck	HDLC Protocol (bug found)

Simulazione

- Mentre il model-checking esplora tutto lo spazio degli stati, la simulazione esplora un solo cammino
- Analogie con il testing del codice
- Può trovare errori, ma non li può trovare tutti
- Si possono generare solo gli stati esplorati, quindi si evita l'esplosione degli stati
- Occorre "accorgersi" che si passa da uno stato già esplorato, oppure si taglia la profondità della simulazione

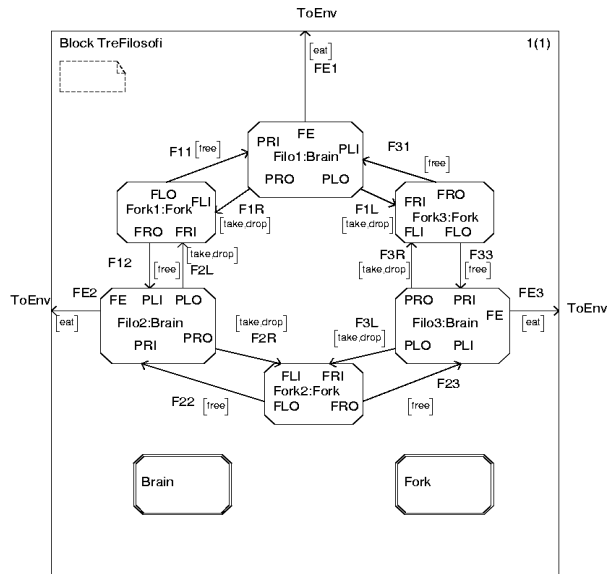


Simulazione

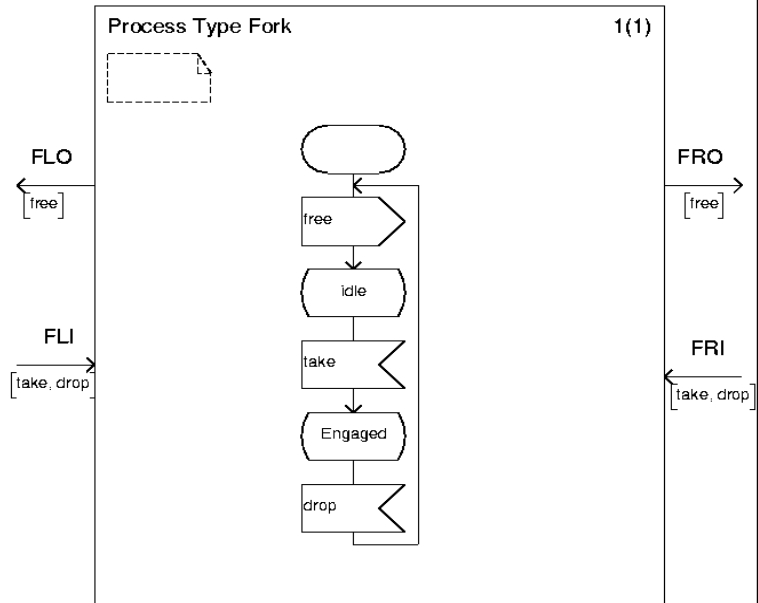
- Simulazione secondo "scenari" (analogie con i casi di test)
- Simulazione casuale
- Simulazione esaustiva breadth first
- Simulazione esaustiva depth first
(in entrambi questi casi si può limitare la profondità della simulazione)
- Simulazione anche dei valori delle variabili (altro elemento di non esaustività)
- Caratterizzata comunque dal fatto che è possibile generare solo gli stati localmente necessari alla simulazione ("on the fly")
- Rappresentazione esplicita degli stati

SDL

Linguaggio standard per la descrizione di sistemi nato in ambito telecomunicazioni (ITU-T)

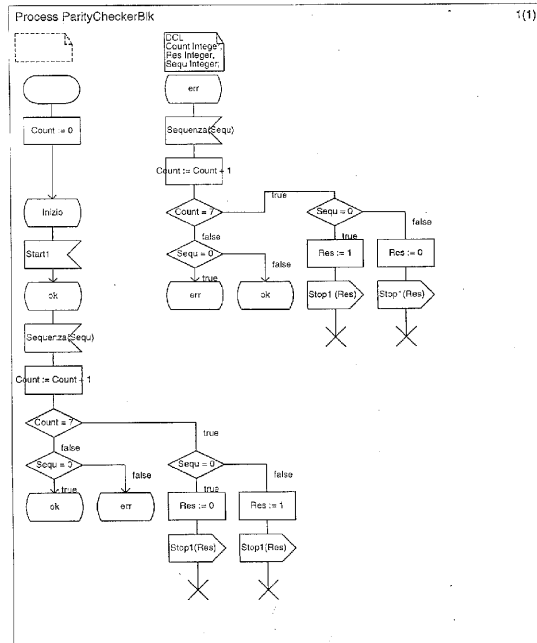


SDL



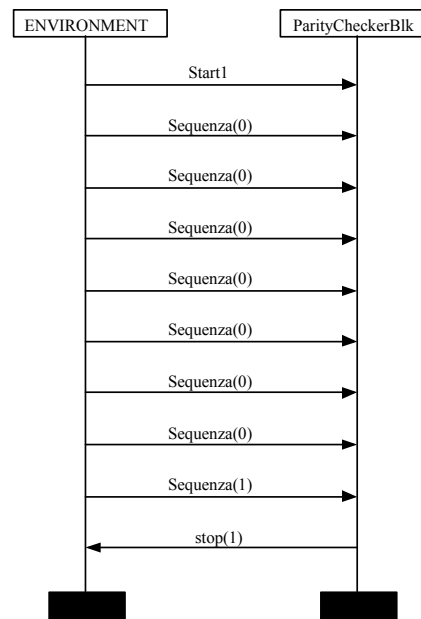
Esempio di verifica mediante simulazione

SDL esprime il modello del sistema,



Message Sequence Chart (MSC)

Permette di descrivere una computazione mediante la storia dei messaggi scambiati tra i processi



Risultati della simulazione

La simulazione “guidata” dallo scenario fornito sotto forma di MSC permette di stabilire se tale scenario è una computazione ammissibile dal modello oppure tale computazione non è possibile. Nell'esempio, la risposta è affermativa.

Si possono osservare gli stati attraversati dalla simulazione, mediante **animazione** della visualizzazione grafica della specifica o di sue porzioni.

Oppure si possono ottenere misure di **copertura** per conoscere la percentuale degli stati attraversati rispetto al totale.

Questa misura di copertura può dare indicazioni sulla “completezza” delle prove effettuate.

È evidente che questa verifica da sola non basta a stabilire la correttezza della specifica, ma può solo servire a rilevare eventuali errori. Si può notare invece che si potrebbero usare tecniche di model checking qualora si riuscisse a codificare in logica temporale la seguente affermazione:

"tutte le sequenze hanno un numero pari di uni se e solo se il risultato è stop(0)"

Questo si può fare usando il μ -calcolo (una logica temporale lineare con operatore di punto fisso) [Koz83], con la seguente formula:

$\mu X.(F (Sequenza(1) \& F (Sequenza(1) \& X)) | (\sim Sequenza(1) \cup Stop (0)))$

IAR Visualstate

- Nato in ambito Bang & Olufsen al fine di progettare interfacce di qualità per i loro prodotti Hi-Fi di alta fascia.
- Include, oltre al Designer,
 - un Validator (che permette di simulare il comportamento del sistema e di animare la visualizzazione grafica del modello)
 - un Verificator che nasconde un potente model checker, e che permette di verificare una serie di proprietà generiche quali l'assenza di deadlock, o l'assenza di conflitti nondeterministici
 - un Prototyper che permette di disegnare un prototipo di interfaccia e di collegarla agli ingressi/uscite per il simulatore
- La simulazione avviene attraverso la generazione del codice dal modello. Lo stesso codice, previa attualizzazione delle parti dipendenti dalla macchina e dal sistema operativo, può essere direttamente impiegato come implementazione (tipicamente su un sistema embedded)

Ilogix Statemate

- Permette la composizione di Statecharts in modo più flessibile rispetto a Visualstate, supportando un processo di scomposizione top-down (adatto per sistemi di più grandi dimensioni)
- Funzionalità di simulazione e animazione, su scenari descritti attraverso MSC (nella forma di Sequence Diagrams di UML) e Timing Diagrams.
- Disponibile un potente model checker, sia con formule "cablate" che con formule "libere"
- Generazione di codice
- Generazione di casi di test
- Prototipazione di interfacce

Telelogic SDT

- Uso di SDL e MSC standard – provvisti di rappresentazione testuale: in teoria e` possibile scambiare specifiche SDLe MSC tra strumenti di vendors diversi. (in pratica il principale concorrente di SDT, ObjectGeode della Verilog, non e` piu` tale, perche' la Telelogic ha acquistato la Verilog -> situazione di monopolio di fatto)
- Simulazione del modello SDL guidata da MSC
- Tecniche avanzate (prese in prestito da Model checking) per la rappresentazione efficiente dello spazio degli stati
- Analisi di copertura dello spazio degli stati esercitata da una simulazione
- Generazione di codice
- Generazione di casi di test in formato TTCN (standard)

Strumenti di verifica (semi-)commerciali

- SMV - model-checker accademico
- (Carnegie Mellon University)
- SPIN - model-checker semi-accademico
- (AT&T Labs)
- NP-Tools - procedura di decisione - commerciale (Prover Technology)
 - Interventi a lato dello sviluppo tradizionale
 - Supporto diretto da parte di esperti esterni
 - (servizio di consulenza da parte del produttore
 - (penetrazione nel contesto industriale limitata)

Sistemi sincroni

- SCADE (Telelogic) – Linguaggio LUSTRE
- Linguaggio ESTEREL (INRIA)
 - Sistemi definiti come reti di macchine a stati finiti sincrone
 - Formalismi adatti per definire sistemi embedded, non distribuiti
 - Il modello del sistema e' molto vicino all'implementazione.
 - Infatti gli strumenti sono in genere equipaggiati di generatori di codice.

Real Time

- Real Time Model Checking:
 - CHRONOS (Verimag)
 - UPPAAL (Univ. of Aarhus)
- Hybrid Model Checking:
 - Hytech (Berkeley)
 - Lo stato del sistema e' in parte definito da un sistema di transizione discreto, in parte definito da un sistema continuo (variabili di stato definite in funzione del tempo, attraverso equazioni differenziali)
- Schedulability Analysis
 - Problemi di esplosione dello spazio degli stati amplificati.
 - In ambito industriale, spesso si ricorre al conteggio delle istruzioni macchina eseguite per garantire la rispondenza a deadline temporali.

Tendenze nei vari settori applicativi

- Ferroviario: *B, Statemate, SDT, NPTools, SMV, SPIN, SCADE*
- Avionico/Spaziale: *SCADE, ESTEREL*
- Automobilistico: *Statemate, SCADE, ESTEREL, HyTECH*
- Telecomunicazioni: *SDT (SDL), SPIN*

Normativa CENELEC

European Committee for Electrotechnical
Standardization

prEN 50128 June 1997

Railway Applications:
Software for Railway Control
and Protection Systems

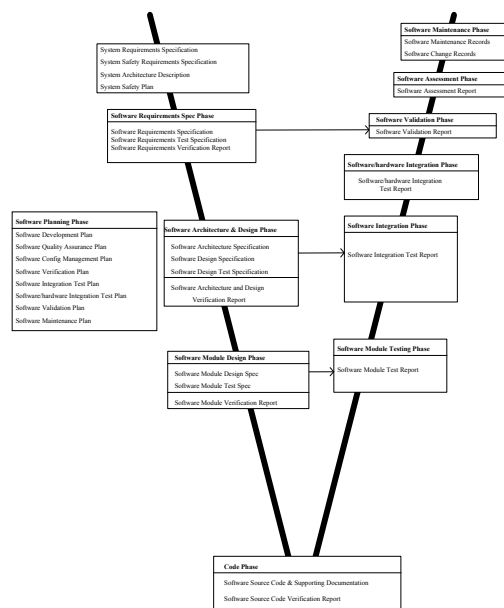
Software safety integrity levels (SIL)

- The required software safety integrity level shall be decided on the basis of the level of risk associated with the use of the software in the system and the system safety integrity level.

<i>Integrity Level</i>	<i>Safety Integrity</i>
4	Very High
3	High
2	Medium
1	Low
0	Non Safety-Related

Modello di ciclo di vita a V

A ogni passo del ciclo di sviluppo corrisponde un passo di validazione



Ruolo dei Metodi Formali nella normativa CENELEC

With each technique or measure in the tables there is a requirement for each software safety integrity level (SWSIL), 1 to 4 and also for the non safety-related level 0. In this version of the document, the requirements for software safety integrity levels 1 and 2 are the same for each technique. Similarly, each technique has the same requirements at software safety integrity levels 3 and 4. These requirements can be:

- 'M' This symbol means that the use of a technique is mandatory.
- 'HR' This symbol means that the technique or measure is Highly Recommended for this safety integrity level. If this technique or measure is not used then the rationale behind not using it should be detailed in the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan.
- 'R' This symbol means that the technique or measure is Recommended for this safety integrity level. This is a lower level of recommendation than an 'HR' and such techniques can be combined to form part of a package.
- '-' This symbol means that the technique or measure has no recommendation for or against being used.
- 'NR' This symbol means that the technique or measure is positively Not Recommended for this safety integrity level. If this technique or measure is used then the rationale behind using it should be detailed in the Software Quality Assurance Plan or in another document referenced by the Software Quality Assurance Plan.
- 'F' This symbol means that the use of this technique is forbidden.

Ruolo dei Metodi Formali nella normativa CENELEC

Table A2 : Software Requirements Specification

TECHNIQUE/MEASURE	Ref	SWSIL 0	SWSIL 1	SWSIL 2	SWSIL 3	SWSIL 4
1. Formal Methods including for example CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z and B	B.30	-	R	R	HR	HR
2. Semi-Formal Methods	D.7	R	R	R	HR	HR
3. Structured. Methodology including for example JSD, MASCOT, SADT, SDL, SSADM, and Yourdon.	B.60	R	HR	HR	HR	HR
Requirements 1. The Software Requirements Specification will always require a description of the problem in natural language and any necessary mathematical notation that reflects the application. 2. The table reflects additional requirements for defining the specification clearly and precisely. One or more of these techniques shall be selected to satisfy the Software Safety Integrity Level being used.						

Table A4: Software Design and Implementation

TECHNIQUE/MEASURE	Ref	SWSILO	SWSIL1	SWSIL2	SWSIL3	SWSIL4
1. Formal Methods including for example CCS, CSP, HOL, LOTOS, OBJ, Temporal Logic, VDM, Z and B	B.30	-	R	R	HR	HR
2. Semi-Formal Methods (incl. Statecharts, FSMs)	D.7	R	HR	HR	HR	HR
3. Structured Methodology including for example JSD, MASCOT, SADT, SDL, SSADM and Yourdon.	B.60	R	HR	HR	HR	HR
4. Modular Approach	D.9	HR	M	M	M	M
5. Design and Coding Standards	D.1	HR	HR	HR	M	M
6. Analysable Programs	B.2	HR	HR	HR	HR	HR
7. Strongly Typed Programming Language	B.57	R	HR	HR	HR	HR
8. Structured Programming	B.61	R	HR	HR	HR	HR
9. Programming Language	D.4	R	HR	HR	HR	HR
10. Language Subset	B.38	-	-	-	HR	HR
11. Validated Translator	B.7	R	HR	HR	HR	HR
12. Translator Proven in Use	B.65	HR	HR	HR	HR	HR
13. Library of Trusted/Verified Modules and Components	B.40	R	R	R	R	R
14. Functional/ Black-box Testing	D.3	HR	HR	HR	M	M
15. Performance Testing	D.6	-	HR	HR	HR	HR
16. Interface Testing	B.37	HR	HR	HR	HR	HR
17. Data Recording and Analysis	B.13	HR	HR	HR	M	M
18. Fuzzy Logic	B.67	-	-	-	-	-
19. Object Oriented Programming	B.68	-	R	R	R	R
Requirements						
1. A suitable set of techniques shall be chosen according to the software safety integrity level.						
2. At software safety integrity level 3 or 4, the approved set of techniques shall include one of techniques 1, 2 or 3, together with one of techniques 11 or 12. The remaining techniques shall still be treated according to their recommendations.						

Table A5: Verification and Testing

TECHNIQUE/MEASURE	Ref	SWSILO	SWSIL1	SWSIL2	SWSIL3	SWSIL4
1. Formal Proof	B.31	-	R	R	HR	HR
2. Probabilistic Testing	B.47	-	R	R	HR	HR
3. Static Analysis	D.8	-	HR	HR	HR	HR
4. Dynamic Analysis and Testing	D.2	-	HR	HR	HR	HR
5. Metrics	B.42	-	R	R	R	R
6. Traceability Matrix	B.69	-	R	R	HR	HR
7. Software Error Effects Analysis	B26	-	R	R	HR	HR
Requirements						
1. For Software Safety Integrity Level 3 or 4, the approved combinations of techniques shall be:- a) 1 and 4 b) 3 and 4 c) 4, 6 and 7 or 2. For Software Safety Integrity Level 1 or 2, the approved technique shall be 1 or 4.						

Table A18: Semi-Formal Methods (D.7)

Referenced by clauses 8 and 10

TECHNIQUE/MEASURE	Ref	SWS ILO	SWS IL1	SWS IL2	SWS IL3	SWS IL4
1. Logic/Function Block Diags	-	R	R	R	HR	HR
2. Sequence Diagrams	-	R	R	R	HR	HR
3. Data flow Diagrams	B.12	R	R	R	R	R
4. Finite State Machines/State Transition Diagrams	B.29	-	R	R	HR	HR
5. Time Petri Nets	B.64	-	R	R	HR	HR
6. Decision/Truth Tables	B.14	R	R	R	HR	HR