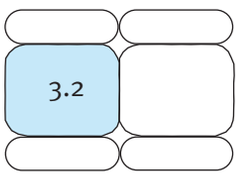
MODEL CHECKING
COS'È E COME SI APPLICA

Alessandro Fantechi
Stefania Gnesi

Il model checking ha dimostrato di essere una tecnologia di successo per verificare la correttezza dei requisiti nella progettazione di un consistente numero di sistemi *real-time*, *embedded* e *safety-critical*. Lo scopo di questo breve articolo è di spiegare come funziona.



1. INTRODUZIONE

Lo sviluppo di metodologie formali per la progettazione, l'analisi e la verifica dei moderni sistemi software, spesso realizzati mediante l'assemblaggio di componenti preesistenti, assume un'importanza cruciale nel fornire un valido sostegno alla loro validazione fin dalle prime fasi del loro sviluppo. Al fine di evitare l'introduzione di difetti di progettazione o l'insorgere di problemi di integrazione, o d'interoperabilità nell'interconnessione dinamica dei componenti software è importante affiancare alla specifica del sistema in progettazione opportune metodologie di verifica.

È prassi comune scrivere i requisiti di un sistema in lingua naturale corredando le descrizioni testuali con diagrammi, *screenshot* e notazioni UML, come per esempio i casi d'uso e diagrammi di sequenza, di classe e di stato.

L'attività di verifica dei requisiti consiste fondamentalmente nel cercare le risposte a una serie di domande quali per esempio:

- ❑ I requisiti soddisfano le esigenze degli utenti? Tutto ciò che è stato descritto corrisponde a ciò che essi vogliono e tutto ciò che gli utenti hanno chiesto è incluso?
- ❑ I requisiti sono scritti chiaramente e non contengono ambiguità? Sono comprensibili?
- ❑ I requisiti sono opportunamente strutturati per facilitare la progettazione e lo sviluppo?
- ❑ I requisiti possono essere utilizzati per definire facilmente i casi di test di accettazione per verificare la conformità del prodotto rispetto ai requisiti stessi?
- ❑ I requisiti sono scritti in modo sufficientemente astratto e ad alto livello in modo da dare sufficiente libertà al progettista e agli sviluppatori di implementare in modo efficiente?

Trovare le risposte a queste domande è un lavoro difficile e non c'è un modo semplice per farlo. Nonostante qualche aiuto possa venire dall'uso di strumenti di modellazione come per esempio UML, il problema di garantire la qualità dei requisiti rimane. E talvolta si introducono ulteriori problemi:

- ❑ Quale notazione usare per quali requisiti?
- ❑ Come garantire che le descrizioni in diverse notazioni siano coerenti tra loro?

Prima ancora di iniziare a scrivere il codice che realizzerà un sistema software, il progettista dovrà esser sicuro che i requisiti che descrivono le funzionalità del sistema stesso non con-

tengano errori. È infatti importante trovare il maggior numero di possibili difetti prima che la fase di codifica venga effettuata perché altrimenti l'eliminazione degli errori quando il prodotto è già stato consegnato potrebbe diventare molto costosa.
Un modo per migliorare la qualità dei requisiti e del progetto è quello di utilizzare strumenti automatici per verificarne la qualità relativamente ad un insieme di caratteristiche considerate importanti al fine di garantire il buon funzionamento del sistema.

Quali strumenti sono più idonei per questo compito?
Un modo potrebbe essere quello di costruire strumenti automatici per verificare i requisiti scritti in linguaggio naturale. Qualcosa in questa direzione è stato fatto, ma sicuramente non è possibile, data la natura intrinseca del linguaggio naturale, realizzare delle analisi esaustive dei requisiti così espressi.
L'introduzione di linguaggi e notazioni formali diventa un passo necessario al fine di poter descrivere i requisiti di un sistema in modo chiaro e non ambiguo e quindi con una semantica ben definita. Partendo da qui sarà possibile sviluppare strumenti automatici per analizzare in modo esaustivo i requisiti stessi.
La progettazione di sistemi software complessi sarà quindi molto facilitata dall'applicazione di metodologie che facciano riferimento in modo diretto o indiretto a descrizioni formali delle applicazioni stesse. Ciò è particolarmente importante per quei sistemi dedicati al controllo di applicazioni critiche, di cui quelle nei settori *automotive*, ferroviario, avionico e spaziale, sono gli esempi più immediati, per i quali le descrizioni formali possono fornire una guida utile per garantire le proprietà necessarie di correttezza, sicurezza, interoperabilità, affidabilità ed efficienza.
La domanda che ora ci dobbiamo porre è quale sia il linguaggio formale più idoneo per definire i nostri sistemi. Non esiste una risposta unica, poiché i fabbisogni per i sistemi in diversi domini applicativi variano notevolmente. Per esempio, i requisiti di un sistema bancario o quelli di un sistema aerospaziale si differenziano molto sia per struttura, complessità, natura dei dati, sia per le operazioni eseguite. Inoltre, nella maggior parte dei sistemi *real-time embedded* o *safety-critical* è necessario avere a disposizione dei linguaggi formali idonei a descrivere aspetti orientati al loro controllo, piuttosto che al trattamento dei dati.
Per quest'ultima classe di sistemi, vari "dialetti" di automi a stati finiti, come macchine a stati finiti (FSM), sistemi di transizione etichettati (LTS), o diagrammi a stati UML, sono ampiamente accettati come una notazione formale chiara, semplice e sufficientemente astratta per esprimere i requisiti.
Negli ultimi venti anni, la ricerca in informatica ha fatto enormi progressi nello sviluppo di strumenti automatici a supporto della verifica dei requisiti di sistemi complessi. Uno degli approcci di maggior successo è quello conosciuto come *model checking*, che permette di automatizzare il processo di verifica di sistemi descritti mediante l'uso di linguaggi formali.
In questo articolo, si darà un'introduzione al *model checking* e si mostrerà come funziona.

2. MODEL CHECKING

La tecnica del model checking, introdotta da Clarke, Emerson e contemporaneamente da Quielle e Sifakis agli inizi degli anni '80, è basata su idee estremamente semplici ed è probabilmente uno dei più significativi avanzamenti della ricerca in Informatica di base di questi ultimi decenni.
Nella verifica tramite *model checking* il sistema sotto analisi viene descritto con un linguaggio formale e successivamente modellato mediante qualche dialetto di automi a stati finiti (nel seguito faremo riferimento alla notazione di base degli automi a stati finiti, che assumeremo come nota). Le proprietà da verificare su tale modello sono rappresentate tramite un linguaggio preciso e non ambiguo, tipicamente in logica temporale, e la loro soddisfacibilità sull'automa a stati finiti che modella il sistema viene verificata in modo efficiente ed automatico. Se le proprietà non sono soddisfatte, una traccia di esecuzione (controesempio) è mostrata al fine di evidenziare perché si ha il fallimento nella verifica.
In sintesi la verifica tramite *model checking*

consiste, dato un automa M modello di un sistema e una formula di logica temporale ϕ, che rappresenta una proprietà che si desidera che M abbia, nel verificare se M soddisfa ϕ.

Come possiamo specificare queste proprietà delle esecuzioni?
Sono state sviluppate diverse logiche temporali a questo scopo. Una logica temporale è un'estensione della logica proposizionale la cui struttura di interpretazione è una successione di istanti di tempo distinti.
Tra le logiche temporali esistenti consideriamo qui la logica LTL (*Linear time Temporal Logic*), una logica temporale che fa riferimento ad una struttura temporale lineare di stati e transizioni, che rappresenta una possibile esecuzione di un sistema modellato mediante un automa. La sintassi della logica LTL è definita a partire dagli operatori della logica booleana proposizionale (che comprende i connettivi logici come AND, OR, NOT, implica ecc.), dove sono stati aggiunti connettivi temporali. Alcuni connettivi temporali tipici di LTL sono rappresentati nella tabella 1, insieme al loro significato informale.
La figura 1 rappresenta graficamente il significato degli operatori temporali descritti nella tabella 1, dando per ogni formula una sequenza di stati e transizioni tra stati che rappresenta un'esecuzione del modello su cui la formula è verificata. Gli stati colorati rappresentano stati in cui le sottoformule della formula considerata sono verificate; la prima sequenza indica che, se non vi sono connettivi temporali, la formula si intende verificata sullo stato iniziale.

3. UN SEMPLICE ESEMPIO

Per mostrare il funzionamento dell'algoritmo di model checking, consideriamo un classico

X ϕ (*next*)	Vera nello stato corrente se la formula ϕ è vera nello stato successivo.
G ϕ (*always*)	Vera nello stato corrente se ϕ è vera in tutti gli stati successivi.
F ϕ (*eventually*)	Vera nello stato corrente se ϕ è vera in almeno uno degli stati successivi.
ϕ_1 U ϕ_2 (*until*)	Vera nello stato corrente se ϕ_2 è vera in uno stato futuro e ϕ_1 è vera in tutti gli stati precedenti.
ϕ_1 P ϕ_2 (*precedes*)	Vera nello stato corrente se ϕ_2 non è vera in uno stato precedente a quello in cui ϕ_1 è vera.

TABELLA 1
Alcuni degli operatori temporali di LTL

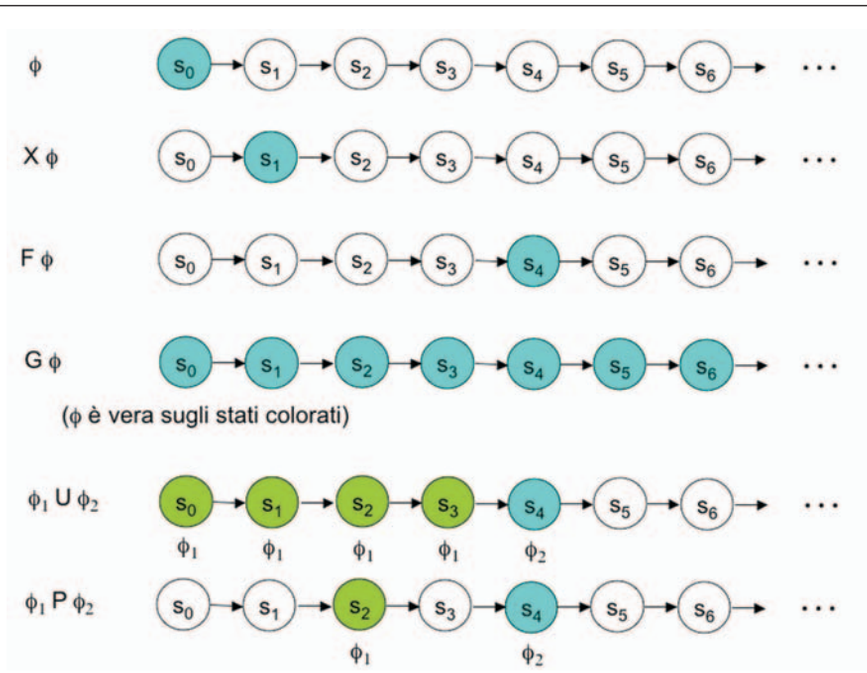


FIGURA 1
Interpretazione degli operatori temporali di LTL

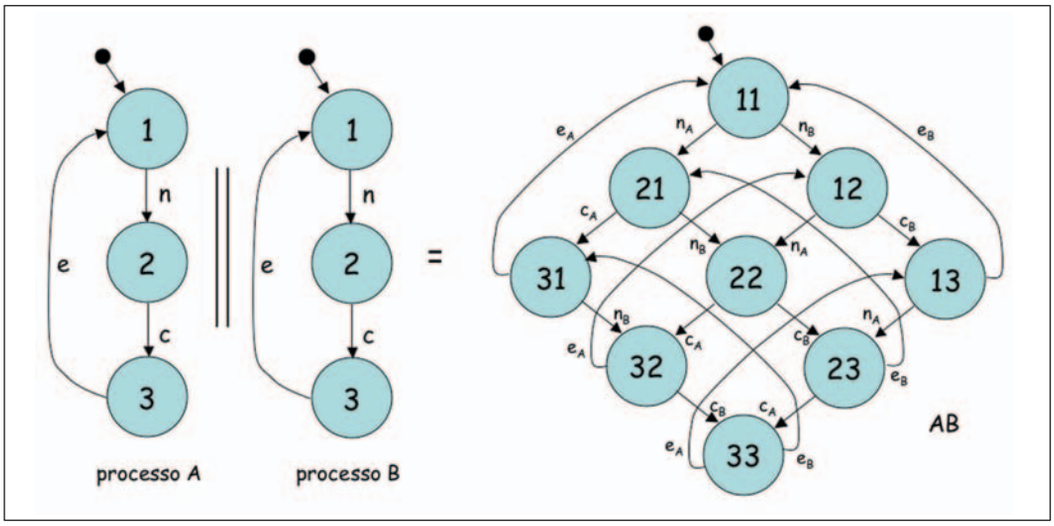


FIGURA 2
Composizione di due processi che accedono ad una risorsa condivisa

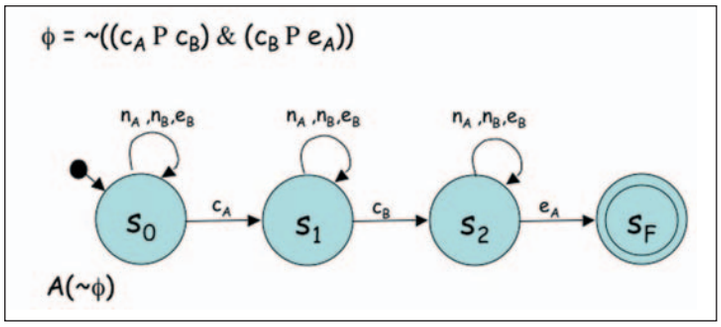


FIGURA 3
Automa della formula da verificare

esempio di sistema concorrente, in cui due processi accedono ad una risorsa condivisa. Per semplificare l'esposizione, consideriamo un semplice modello del funzionamento di ognuno dei due processi, di cui si osservano solamente le principali transizioni di stato:

- una transizione, ***n***, che modella la computazione che viene svolta senza accedere alla risorsa;
- una transizione, ***c***, che modella l'ingresso nella "regione critica", in cui si accede alla risorsa condivisa,
- una transizione, e, che modella il rilascio della risorsa condivisa (si vedano i due automi a sinistra nella Figura 2).

Se eseguiamo in parallelo i due processi A e B, assumendo le transizioni come atomiche, l'automa a stati finiti che rappresenta il comportamento complessivo del sistema dei due processi è l'automa AB, a destra nella figura 2 (c_A e c_B rappresentano la transizione ***c*** eseguita dal processo A e B rispettivamente, ed analogamente per le transizioni n ed e).

Supponiamo che ci interessi provare la proprietà di *mutua esclusione*, ovvero che non vi è interferenza sull'accesso alla risorsa condivisa. Possiamo esprimere questa proprietà dicendo che se il processo A accede per primo alla risorsa, il processo B non deve accedere prima che A abbia rilasciato la risorsa. Ovvero, non devono esistere sequenze di transizioni in cui c_A precede c_B e c_B precede e_A.

In logica temporale LTL questo si esprime come:

$$\phi = \sim(c_A \, P \, c_B \,\&\, c_B \, P \, e_A)$$

Per specificare completamente la mutua esclusione dovremmo poi esprimere anche la formula simmetrica, con A e B scambiati, ma ci limitiamo a questa parte della proprietà.

Per verificare questa proprietà si procede definendo un automa $A(\sim\phi)$ che generi tutte le sequenze di transizione non ammesse dalla suddetta formula (Figura 3). L'intersezione dell'automa AB con l'automa $A(\sim\phi)$ ci fornisce un automa che genera le sequenze che sono riconosciute da entrambi gli automi. Se la proprietà fosse verificata, questa intersezione ci dovrebbe quindi dare il linguaggio vuoto. Questo non è vero in questo caso, in quanto la sequenza $n_A \, n_B \, c_A \, c_B \, e_A$ appartiene all'intersezione. Questa sequenza (Figura 4) viene detta "controesempio", in

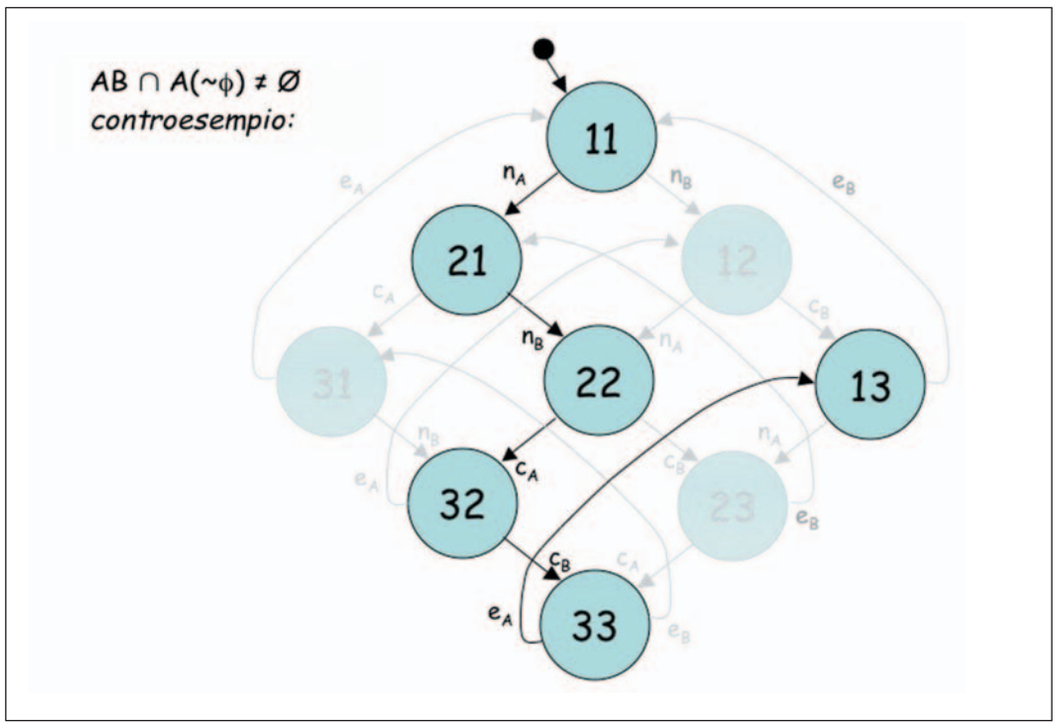


FIGURA 4
Cammino di controesempio sull'automa AB

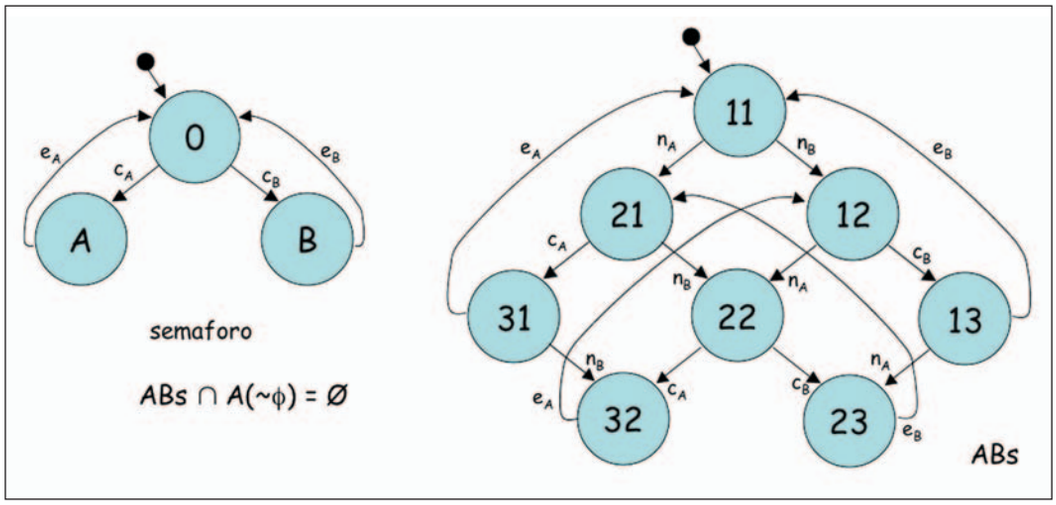


FIGURA 5
Comportamento del sistema che usa un semaforo

quanto dimostra la non veridicità della formula per il sistema AB (in questo caso non è l'unico controesempio ricavabile). Il controesempio può servire a diagnosticare il motivo per cui il sistema non soddisfa la proprietà richiesta: in questo caso ci si è chiaramente dimenticati di impedire l'accesso del processo B alla risorsa quando il processo A l'ha già acquisita.
A questa dimenticanza si può ovviare con il classico meccanismo del semaforo, che presenta il comportamento descritto dall'automa a sinistra nella figura 5. Sincronizzando i processi A e B con il semaforo, otteniamo l'automa ABs, rappresentato a destra nella figura 5. Ripetendo il procedimento di model checking si vede che questa volta l'intersezione tra A(~ϕ) e ABs è nulla, e quindi la formula è soddisfatta.
Si noti che i passaggi che sono stati effettuati in questo esempio sono tutti formalizzabili come operazioni su automi e sono implementabili con opportuni algoritmi. Questo significa che il procedimento esemplificato può essere codificato in un algoritmo che dimostra formalmente che la proprietà vale o,

in caso contrario, fornisce un controesempio. Questo algoritmo, modificato per lavorare su *automi di Büchi*, ovvero automi che riconoscono sequenze infinite, è sostanzialmente quello utilizzato nel Model Checker SPIN. L'algoritmo di model checking di Clarke ed Emerson agisce invece visitando gli stati dell'automa ed etichettandoli progressivamente con le sottoformule della formula da verificare, formule espresse nella logica temporale CTL.
Entrambi questi algoritmi presentano una complessità computazionale lineare con la dimensione dello spazio degli stati del modello.

4. LIMITI DEL MODEL CHECKING

L'esempio della mutua esclusione mostra come l'algoritmo di model ckecking si applichi ad un modello a stati ottenuto tramite la composizione dei modelli di più processi. Come l'esempio mostra, in genere il numero degli stati del sistema complessivo cresce, quando il sistema è composto di più processi, in modo esponenziale con il numero dei processi coinvolti, perché lo spazio degli stati può essere il prodotto degli spazi degli stati di ogni processo. Questo fenomeno, chiamato "esplosione dello spazio degli stati", limita fortemente la capacità degli algoritmi di model ckecking citati a lavorare su esempi di sistemi "reali", fermandosi intorno al milione di stati, principalmente a causa della dimensione dello spazio di memoria necessario per rappresentare lo spazio degli stati.
Le limitazioni degli algoritmi di model checking dovute all'esplosione dello spazio degli stati sono state affrontate con la tecnica introdotta da McMillan che permette la rappresentazione simbolica degli stati per mezzo degli *Ordered Binary Decision Diagrams* (OBDD). In questa tecnica lo spazio degli stati non viene rappresentato esplicitamente memorizzando tutti gli stati e le transizioni, ma implicitamente (o simbolicamente) mediante un insieme di funzioni binarie, a loro volta memorizzate in modo compatto tramite OBDD, arrivando a poter trattare sistemi da 10^{23} a 10^{120} stati.
Questa tecnica ha quindi posto le basi per una vasta diffusione del model checking, soprattutto nell'ambito della verifica dell'hardware, tanto da valere ai pionieri Clarke, Emerson e Sifakis il prestigioso ACM Turing Award del 2007.
La ricerca sul model checking si è da allora sviluppata enormemente e varie tecniche sono state studiate per migliorarne ancora le prestazioni e tentare di ridurre il fenomeno dell'esplosione degli stati. Una possibilità è quella di generare e memorizzare solo lo spazio degli stati necessario per verificare la proprietà in questione.
In particolare, nel *local model checking* (anche noto come *on-the-fly model checking*) si generano, sulla base della struttura della formula, solo gli stati che effettivamente servono a verificare la formula (in alcuni casi però potrebbe essere necessario generare l'intero spazio degli stati).
Nel *Bounded Model Checking* (BMC) invece lo spazio degli stati viene generato fino ad una profondità fissata: il risultato del BMC sarà quindi ternario: la formula è verificata, la formula non è verificata, la profondità non è sufficiente per verificare. Nel terzo caso, si può decidere di aumentare la profondità. Il BMC sta vivendo un momento di popolarità da quando si è dimostrato che il problema del model checking LTL su uno spazio degli stati di profondità limitata si può esprimere efficientemente come un problema di soddisfacibilità (SAT), problema che, pur essendo in generale NP-completo, viene affrontato con successo da algoritmi molto efficienti recentemente sviluppati.

5. MODEL CHECKING IN PRATICA

La disponibilità di strumenti di model checking efficienti e capaci di lavorare su spazi degli stati di grandi dimensioni ne ha favorito, a partire dalla metà degli anni novanta, la diffusione anche in ambito industriale.
I due model checker di origine accademica più famosi sono SMV, sviluppato presso la Carnegie Mellon University, come model checker basato su rappresentazione BDD per la logica CTL e di cui ne esiste una versione reingegnerizzata, NuSMV, sviluppata dalla Fondazione Bruno Kessler di Trento, e SPIN, sviluppato come model checker per la logica temporale lineare (LTL), presso i Bell

Labs sotto la direzione di Gerard Holzmann. Dopo il famoso caso in cui Intel dovette nel 1994 ritirare dal mercato i nuovi chip Pentium, affetti da un guasto di progetto dell'algoritmo di divisione in virgola mobile, rilevato anche tramite model checking, nel settore della produzione di chip questa tecnica si è ormai affermata come metodologia insostituibile di prova della corretta progettazione utilizzando vari model checker proprietari, costruiti spesso in casa dalle ditte produttrici.

Diversa e più variegata è la situazione dell'utilizzo del model checking per la verifica della correttezza del software. In effetti, la produzione dell'hardware digitale ha sempre fatto riferimento a macchine a stati finiti per rappresentare convenientemente il comportamento richiesto da un dispositivo. L'applicazione del model checking, una volta superate le limitazioni di dimensione, ha quindi trovato facile presa tra gli addetti. Nel caso della produzione del software, nonostante l'esecuzione di un programma possa essere sempre in teoria modellata come una macchina a stati, la corrispondenza tra codice e modello non è immediata. Inoltre, in molti casi il software ha, almeno da un punto di vista teorico, un numero di stati infinito o, al meglio, enormemente grande. Se in generale il problema del model checking è indecidibile su programmi a stati infiniti, basta un semplice programma in cui due variabili intere a 32 bit, possono assumere valori indipendenti tra loro e non limitabili a priori, a produrre uno spazio di 2^{64} stati.

Il model checking ha quindi stentato a farsi strada nella produzione del software, dove il testing viene ancora visto come la principale attività di verifica della correttezza del codice, pur non possedendo le doti di esaustività che costituiscono il principale vantaggio del model checking. Se quindi, per il software di utilizzo comune questa tecnica non si è affermata, è stata invece presa in seria considerazione, anche se con un certo ritardo rispetto all'hardware, nel settore della produzione del software di sistemi critici, dove è necessario garantire la massima affidabilità raggiungibile mantenendo i costi delle attività di verifica al di sotto di una soglia accettabile.

6. MODEL BASED DEVELOPMENT

Nell'applicazione del model checking alla produzione del software di sistemi critici, si possono individuare due tendenze principali: la prima, che possiamo considerare più matura e diffusa, si situa all'interno dei metodi di sviluppo basati sulla modellazione formale dei requisiti (*model based*).

In questo approccio lo sviluppo del software avviene a partire da modelli del sistema, attraverso passi di raffinamento e traduzione, anche supportati da appositi strumenti (simulatori, generatori di codice ecc.). La definizione di modelli accurati del funzionamento del sistema, prima dello sviluppo del codice, ha effetti positivi sulla possibilità di rilevare errori di progettazione, quando la loro correzione ha un costo molto minore. La rilevazione di errori può avvenire tramite simulazione del modello, o tramite, appunto, la sua verifica formale. Il modello può poi direttamente servire per generare automaticamente il codice tramite appositi strumenti: questa possibilità è interessante per la realizzazione di prototipi - consentendo di concentrare gli sforzi di progettazione e verifica nella specifica dei modelli - ma viene sempre più diffusamente usata anche per la produzione finale del codice.

Diversi strumenti commerciali di ausilio alla progettazione *model based*, come Simulink/Stateflow di Matlab, o SCADE, contengono moduli che implementano varie versioni di algoritmi di model checking. In altri strumenti di supporto a questo paradigma di sviluppo (come IAR Visualstate), il motore di model checking è nascosto allo sviluppatore, permettendogli di provare proprietà pre-cablate sui modelli sviluppati.

Secondo questa tendenza, l'applicazione del model checking avviene per verificare i modelli, lasciando al generatore di codice il compito di derivarne un codice corretto.

Una delle esperienze più interessanti in questo ambito è quella della Rockwell Collins, operante nel settore avionico, dove sono stati usati come ambienti di progetto *model based* sia Simulink/Stateflow che SCADE, e sono stati sviluppati traduttori per poter applicare ai modelli prodotti diversi model checkers, tra cui SMV e NuSMV. Uno dei risul-

tati più interessanti di questa esperienza è la conferma che il model checking è più efficace del *testing* nel trovare errori: in un esperimento di verifica fatto con due team indipendenti sullo stesso sottosistema, quello che usava model checking ha trovato 12 errori, mentre il team di *testing*, nonostante abbia utilizzato più della metà del tempo speso nell'attività di model checking dall'altro team, non ha potuto rilevare alcun errore.

7. SOFTWARE MODEL CHECKING

La seconda tendenza, che possiamo considerare ancora in via di sviluppo, è quella dell'applicazione diretta delle tecniche di verifica al codice prodotto, e va sotto il nome di "Software Model Checking".
In questo caso, l'approccio è opposto a quello visto in precedenza: si parte dal codice, in qualunque modo sviluppato, e se ne ricava lo spazio degli stati. Essenziale è in questo approccio l'adozione di potenti tecniche di astrazione, che siano in grado di ridurre sostanzialmente la dimensione dello spazio degli stati, senza incidere sulla soddisfacibilità o meno delle proprietà da provare.
Il lavoro pionieristico all'applicazione diretta del model checking al codice è stato fatto alla NASA dalla fine degli anni '90 adottando due diverse strategie: la prima, traducendo il codice nel linguaggio di input di un model checker esistente, per esempio traducendo in PROMELA, linguaggio di input per SPIN. In secondo luogo, attraverso lo sviluppo di model checker *ad hoc* che prendano direttamente i programmi come input. In entrambi i casi, vi è la necessità di estrarre dal codice un modello a stati finiti, anche quando esso abbia teoricamente un numero infinito di stati, e comunque con l'obiettivo di diminuire lo spazio degli stati ad una dimensione gestibile. L'astrazione è quindi la chiave per il model checking del software ed ha lo scopo di eliminare i dettagli irrilevanti al fine della verifica delle proprietà.
In questi ultimi anni sono stati sviluppati un certo numero di software model checker che prendono come input il codice. Alcune caratteristiche comuni di questi sono l'uso nella maggior parte dei casi di una rappresentazione esplicita spazio degli stati, comunque generata *on-the-fly*, e l'adozione di un motore di astrazione che si basa su CEGAR (*Counterexample Guided Abstraction Refinement*), un procedimento iterativo che permette di raffinare l'astrazione del modello fino a trovare quella che permette di verificare (o falsificare) effettivamente la formula, eliminando i falsi positivi o falsi negativi che possono risultare dalla verifica fatta sul sistema astratto.
Il più noto tra i software model checker è JavaPathFinder, realizzato in un progetto sviluppato dalla Nasa per verificare i programmi Java. JavaPathFinder comprende il traduttore Bandera da Java Bytecode ad un certo numero di noti model checkers.
Infatti, lo strumento fornisce una Java Virtual Machine che non esegue normalmente il bytecode, ma controlla direttamente su questo le proprietà da verificare. JavaPathFinder è stato usato per verificare software installati su sonde spaziali, in particolare si riporta l'individuazione e la correzione durante il volo di un errore nel software a bordo della sonda Deep Space DS-1.
Ci riferiamo alla tabella 2 per un elenco di alcuni software model checkers di libera distribuzione. In particolare tra questi vogliamo citare SLAM, sviluppato da Microsoft e utilizzato per verificare che i *driver* di Windows rispettino le convenzioni API.
In entrambe le tendenze (*Model Based Verification e Software Model Checking*) una difficoltà intrinseca, che talvolta frena l'adozione del model checking nella produzione del software, è quella relativa all'espressione delle proprietà desiderate in logica temporale. A questa difficoltà alcuni strumenti di model checking ovviano fornendo all'utente linguaggi più amichevoli ma meno espressivi che propongono proprietà di default "built-in" nel sistema oppure suggeriscono tramite dei template come le proprietà possono essere scritte.

7. CONCLUSIONI

In questo articolo ci siamo limitati a descrivere sommariamente la tecnica del model ckecking e le sue applicazioni alla verifica del software, che si stanno via via consolidando

Strumenti di model checking	Linguaggio di descrizione dei modelli	Linguaggio di specifica delle proprietà
SMV- NuSMV http://www.cs.cmu.edu/~modelcheck/smv.html http://www.kenmcmil.com/smv.html http://nusmv.fbk.eu/	Reti di macchine a stati che comunicano attraverso variabili condivisa	CTL
SPIN http://spinroot.com/spin/whatispin.html	PROMELA	PLTL
Model checker per sistemi temporizzati		
KRONOS http://www-verimag.imag.fr/~tripakis/openkronos.html	Automi temporizzati	Timed CTL
UPAALL http://www.uppaal.com/	Automi temporizzati	Timed CTL
Model checker probabilistici		
PRISM http://www.prismmodelchecker.org/	Catene di Markov a tempo discreto e continuo, processi di decisione markoviani	PCTL, CSL, LTL, PCTL*,
MRMC http://www.mrmc-tool.org/trac/	Modelli di Markov con ricompensa, a tempo discreto e a tempo continuo	PCTL, CSL, PRCTL, CSRL
Software model checkers		
BLAST http://mtc.epfl.ch/software-tools/blast/index-epfl.php CPAChecker http://cpachecker.sosy-lab.org/ SLAM http://research.microsoft.com/en-us/projects/slam/	Programmi C	Annotazioni C
CBMC http://www.cprover.org/cbmc/	Programmi ANSI-C e C++	Asserzioni C
JAVAPATHFINDER http://javapathfinder.sourceforge.net/	Java™ bytecode	Annotazioni JPF, API di verifica

TABELLA 2

Alcuni dei model checker di pubblico dominio

specialmente in determinati settori della produzione del software.
L'attività di ricerca su questa tecnica è ancora molto attiva e promette un suo più vasto utilizzo in vari ambiti dello sviluppo di sistemi informatici. Tra le direzioni di ricerca più seguite negli ultimi anni possiamo citare quella sui sistemi temporizzati, quali i sistemi *real-time*, in cui le proprietà di interesse quantificano con esattezza gli intervalli di tempo in cui azioni e reazioni del sistema debbano avvenire, e quella del model checking probabilistico, dove si vuole valutare quale sia la probabilità che un modello goda di una certa proprietà, aprendo la strada all'utilizzo del model checking nella valutazione quantitativa delle caratteristiche di un sistema.

Bibliografia

[1] Clarke Edmund M., Emerson E. Allen: *Design and synthesis of synchronization skeletons using branching time temporal logic*. Proc. of Workshop in Logics of Programs. 1981.

[2] Quielle Jean-Pierre, Sifakis Joseph: *Specification and verification of concurrent systems in CESAR*. In: Proc. of Symposium on Programming 1982, p. 337-351.

[3] Clarke Edmund M., Emerson E. Allen, Sistla A. Prasad: Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, Vol. 8, n. 2, 1986, p. 244-263.

[4] Clarke Edmund M., Grumberg Orna, Peled Doron: *A Model Checking*. Cambridge, MA: MIT Press, 1999.

[5] Baier Christel, Katoen Joost-Pieter: Principles of model checking. MIT Press, Vol. I-XVII, 2008, p. 1-975.

[6] Holzmann Gerard J.: *The SPIN Model Checker*. Addison-Wesley Professional, 2003.

[7] Miller Steven P., Whalen Michael W., Cofer Darren D.: Software model checking takes off. *Communication of ACM*, Vol. 53, n. 2, 2010, p. 58-64.

Stefania Gnesi è dirigente di ricerca presso il CNR-ISTI ed è responsabile del gruppo di Metodi Formali e Tool dell'ISTI. È stata coordinatrice del gruppo di lavoro dell'ERCIM su Fomal Methods for Industrial Critical Systems. È attualmente vicepresidente dell'associazione Formal Methods Europe. Ha coordinato vari progetti nazionali su metodi e strumenti formali per l'analisi e verifica di sistemi software e ha partecipato a diversi progetti Europei. Svolge attività didattica nel Corso di Laurea Magistrale in Informatica dell'Università di Firenze tenendo lezioni su "Metodi e Strumenti per l'Analisi e la Verifica".
E-mail: stefania.gnesi@isti.cnr.it

Alessandro Fantechi è Professore Ordinario presso la Facoltà di Ingegneria dell'Università di Firenze, dove insegna Informatica Industriale per il Corso di Laurea in Ingegneria Informatica, e Elementi di Software Dependability per il Corso di Laurea Magistrale. I principali interessi di ricerca vertono su applicazioni di metodi formali di specifica e verifica, temi sui quali ha intessuto numerose collaborazioni con aziende produttrici di sistemi safety-critical. Dal Novembre 2008, è coordinatore dello Working Group "Formal Methods for Industrial Critical Systems" del consorzio Europeo ERCIM.
E-mail: fantechi@dsi.unifi.it