Introduzione alla verifica automatica

Sistemi digitali

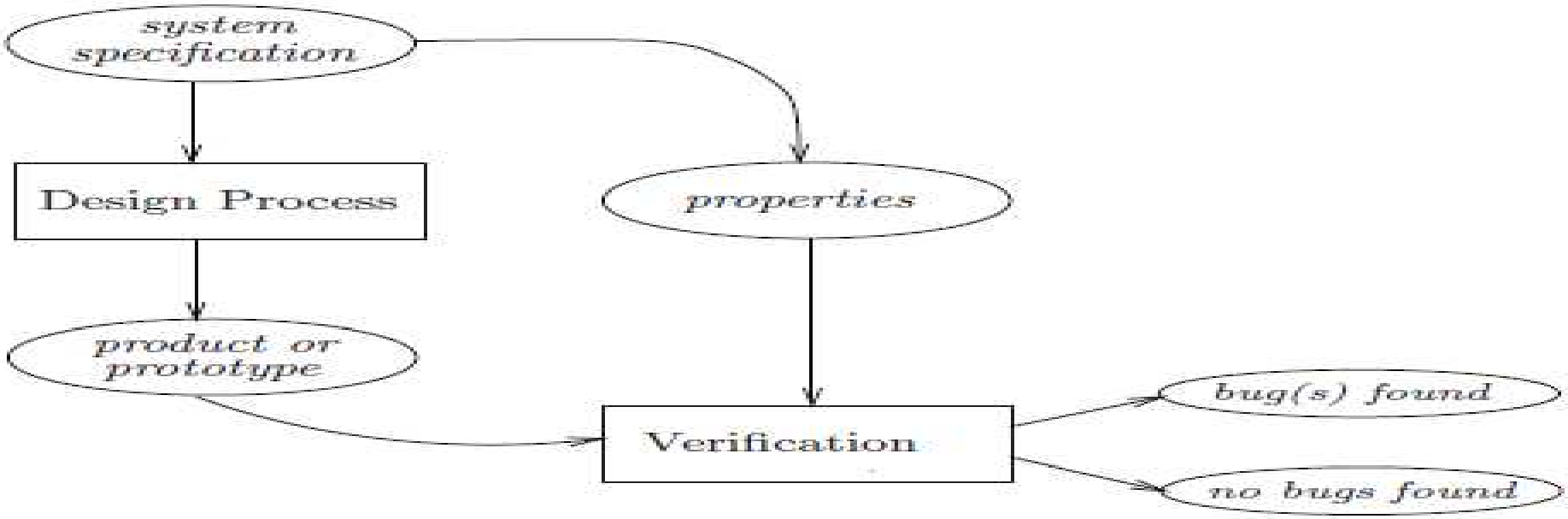
- Utilizzati in quasi tutte le attività umane
- Complessità elevata
  - semplici sistemi hanno milioni di linee di codice
- Tempi di realizzazione sempre più ristretti
  - Concorrenza e contraffazioni spingono a rilasciare nuovi sistemi il prima possibile
- Affidabilità è un requisito primario:
  - bug possono essere molto onerosi
  - difficile da ottenere

Correttezza

- Non è possibile provare un sistema *corretto in maniera assoluta*
- Possiamo solo provare che:
  un sistema ha/non ha una proprietà specificata
- la decisione se ciò costituisce “correttezza” deve stabilirlo una persona
- Dunque,
  - un sistema è corretto se soddisfa i requisiti di progettazione (specifiche)

    (assenza di errori rispetto alle specifiche)

Verifica di correttezza



- Passo importante:
  - specificare le proprietà (requisiti) in maniera corretta
  - ottenere il modello del sistema in maniera accurata

Proprietà tipiche

- alcuni requisiti sono standard:
  - un sistema (ad es. un SO) non dovrebbe andare in deadlock
  - nessun processo dovrebbe essere in grado di impedire ad un altro di accedere ad una risorsa condivisa (lock out)
  - nessuna asserzione esplicitamente enunciata in un programma dovrebbe fallire
- molti sono invece specifici di un applicazione:
  - invarianti di sistemi, asserzioni di processi
  - *requisiti di effettivo progresso*
  - terminazione corretta
  - relazioni causali e temporali sugli stati
    - ad es. ogni richiesta alla fine deve essere esaudita
  - assunzioni di fairness,
    - ad es. riguardo allo scheduling di processi
  - – etc.

Scelta del modello

- un buon modello è un'*astrazione della realtà*
  - deve avere meno dettagli del sistema da modellare
  - il livello di dettaglio è selezionato in base alla rilevanza rispetto alle specifiche di correttezza
  - l'obiettivo è di guadagnare potere di analisi riducendo il dettaglio

Scelta del modello

- lo scopo di un modello è spiegare e consentire previsioni
  - *se non consente di fare nè l'una nè l'altra cosa in quanto approssima troppo, allora non è un buon modello*
- un modello è uno strumento di design
  - spesso ci sono diverse versioni, focalizzate su aspetti differenti, e può diventare gradualmente più accurato senza aumentare il dettaglio *(*accuratezza != dettaglio)

Costruire modelli per la verifica

- Per un dato sistema, vogliamo mantere separati la realizzazione e le specifiche delle proprietà di correttezza
- abbiamo bisogno di due notazioni/formalismi
  - uno per specificare i comportamenti (system design)
  - uno per specificare i requisiti (correctness properties)
- i due tipi di enunciati costituiscono un modello di verifica
- un model-checker può:
  - verificare che un comportamento (design) è logicamnete consistente con un requisito (proprietà)
  - il formalismo deve essere definito in modo che possiamo garantire la decidibilità di ogni proprietà che possiamo enunciare per ogni sistema che possiamo specificare

Panorama metodi di verifica

- Esistono diverse metodologie per verificare la correttezza dei sistemi:
  - Testing
  - Prove di correttezza
  - Sintesi
  - Timing analysis
  - Equivalence checking
  - Simulazione
  - Emulazione
  - Model-checking

Verifica: testing (1)

- verifica sia per software che hardware
- metodo dinamico: viene eseguito il sistema
  - occorre attendere la prima release
- la generazione e l'esecuzione dei test possono essere automatizzate
- il controllo dei risultati del testing è invece difficile da automatizzare
  - spesso richiede l'intervento umano

Verifica: testing (2)

- testing esaustivo di tutte le possibili esecuzioni non è fattibile
  - di solito si considera soltanto un piccolo numero di esecuzioni
  - efficace per error detection, non per dimostrare correttezza
  - difficile determinare quando è sufficiente
- generale: si applica a tutti i tipi di sistema
- il costo del testing nei progetti software è stimato tra il 30% e il 50% del costo totale

Verifica: prove di correttezza

- Solo verifica del software
- svolto da persone esperte di prove formali: preferibilmente non coinvolte nel processo di sviluppo
- metodo statico: ispezione del codice manuale, nessuna esecuzione
- in grado di scoprire dal 31 al 93% degli errori (stimato in in media al 60%)
- alcuni errori impercettibili (tipicamente aspetti di concorrenza o difetti legati alla logica dell'algoritmo) sono difficili da individuare
- individua classi di errori differenti rispetto al testing:
  - solitamente sono utilizzati insieme

Verifica: analisi strutturale

- verifica hardware
- metodo statico
- diverse tecniche:
  - sintesi: circuiti corretti sono sintetizzati rispetto a delle specifiche di alto livello
  - timing analysis: calcolo dei tempi attesi in un circuito digitale senza simulazione
  - equivalence checking: prova formale che due rappresentazioni di un circuito esibiscono gli stessi comportamenti
  - …..

Verifica: emulazione e simulazione

- tecniche dinamiche per verifica di hardware
- emulazione:
  - come il testing ma su un emulatore del sistema sotto analisi
  - emulatore: un circuito riconfigurabile che può essere programmato per comportarsi come il circuito considerato
- simulazione:
  - come il testing ma su un prototipo del sistema
  - è il più utilizzato strumento di verifica per l'hardware
- stessi vantaggi e svantaggi del testing

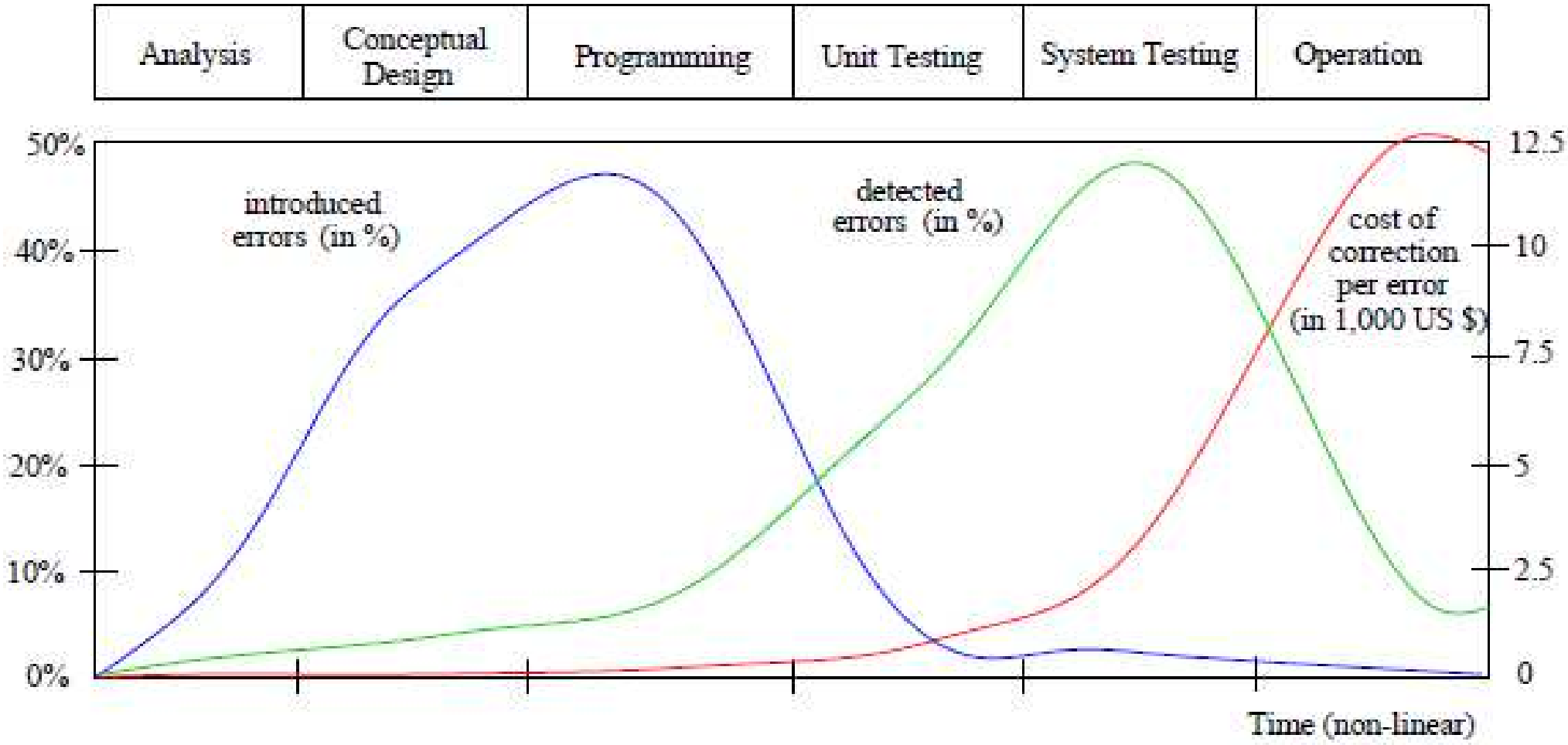
Verifica: model checking

- verifica sia per software che hardware
- metodo statico esaustivo: esplora tutte le esecuzioni di un modello del sistema
- completamente automatico, testa se il modello soddisfa la specifica, e in caso di risposta negativa genera un contro-esempio (traccia d'errore)
- può provare la correttezza del sistema
- aggiunge al testing e alla simulazione in termini di individuazione di bugs

Osservazioni

- Fatto 1: nella progettazione ed implementazione dei sistemi, sono investite più risorse nella validazione che nella costruzione
- Fatto 2: correggere un errore quando il sistema non è stato ancora realizzato costa di meno
- Fatto 3: molti errori logici sono in genere già presenti sin dal primo prototipo
- E' auspicabile l'utilizzo di tecniche di verifica sin dalle prime fasi della progettazione
- Il model-checking utilizza un modello e quindi può essere utilizzato non appena è disponibile

Catching bugs: the sooner, the better



Errori dovuti alla concorrenza

- Difficili da individuare senza una esplorazione esaustiva delle computazioni
- Impossibile da verificare a mano con prove di correttezza
- Semplici programmi hanno un numero di esecuzioni elevatissimo, dovuto ai possibili interleaving (interfogliamento) delle esecuzioni dei singoli processi

Un semplice programma concorrente

```
int   x,  y,  r;
int  *p, *q, *z;
int **a;

thread_1(void)        /* initialize p, q, and z */
{
    p = &x;
    q = &y;
    z = &r;
}
thread_2(void)        /* swap contents of x and y */
{
    r = *p;
    *p = *q;
    *q = r;
}
thread_3(void)        /* access z via a and p */
{
    a = &p;
    *a = z;
    **a = 12;
}
```

3 thread asincrone
con variabili condivise

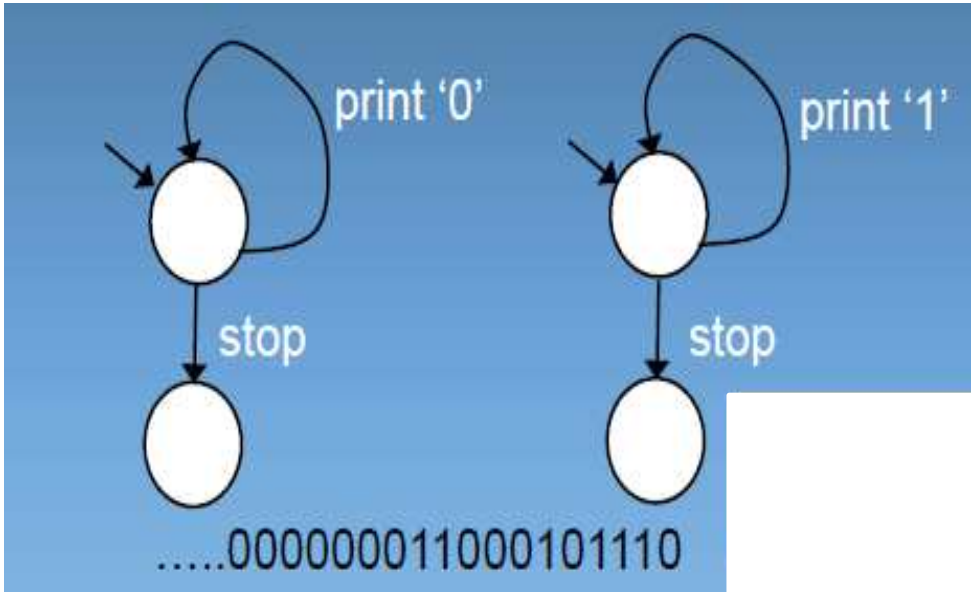
Computazioni

- Assenza di sincronizzazioni
- Comportamento globale si ottiene interfogliando il comportamento di ogni thread
  - Ci sono 1680 modi di interfogliare 3 blocchi di 3 istruzioni ciascuno
    - ogni istruzione occupa un posto da 1 a 9
    - le istruzioni di ogni thread mantengono l'ordine relativo

Due automi finiti

- Un automa stampa un numero arbitrario di “0” e poi termina
- L’altro stampa un numero arbitrario di “1” e poi termina

- I processi corrispondenti sono asincroni
- Ogni sequenza di “0” e “1” può essere generata



Tipico errore difficile da individuare con testing e simulation

- Programma concorrente:

proc Inc = while true do if *x < 200 then x := x + 1 fi od*

proc Dec = while true do if *x > 0 then x := x − 1 fi od*

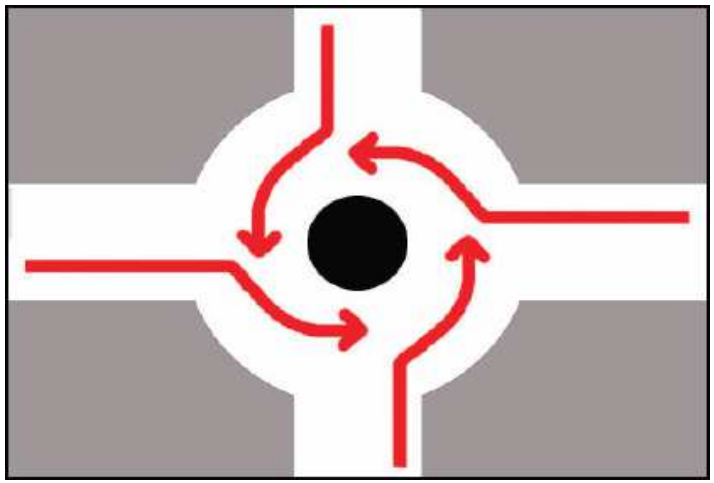
proc Reset = while true do if *x = 200 then x := 0 fi od*

- *Vale l'invariante $x \in [0,200]$?*
  - *NO!*
  - *Controesempio: quando x=200, Dec testa x; allora Reset assegna x con 0; quindi Dec decrementa x. In definitiva, x=-1!*

I sistemi concorrenti sono difficili da progettare

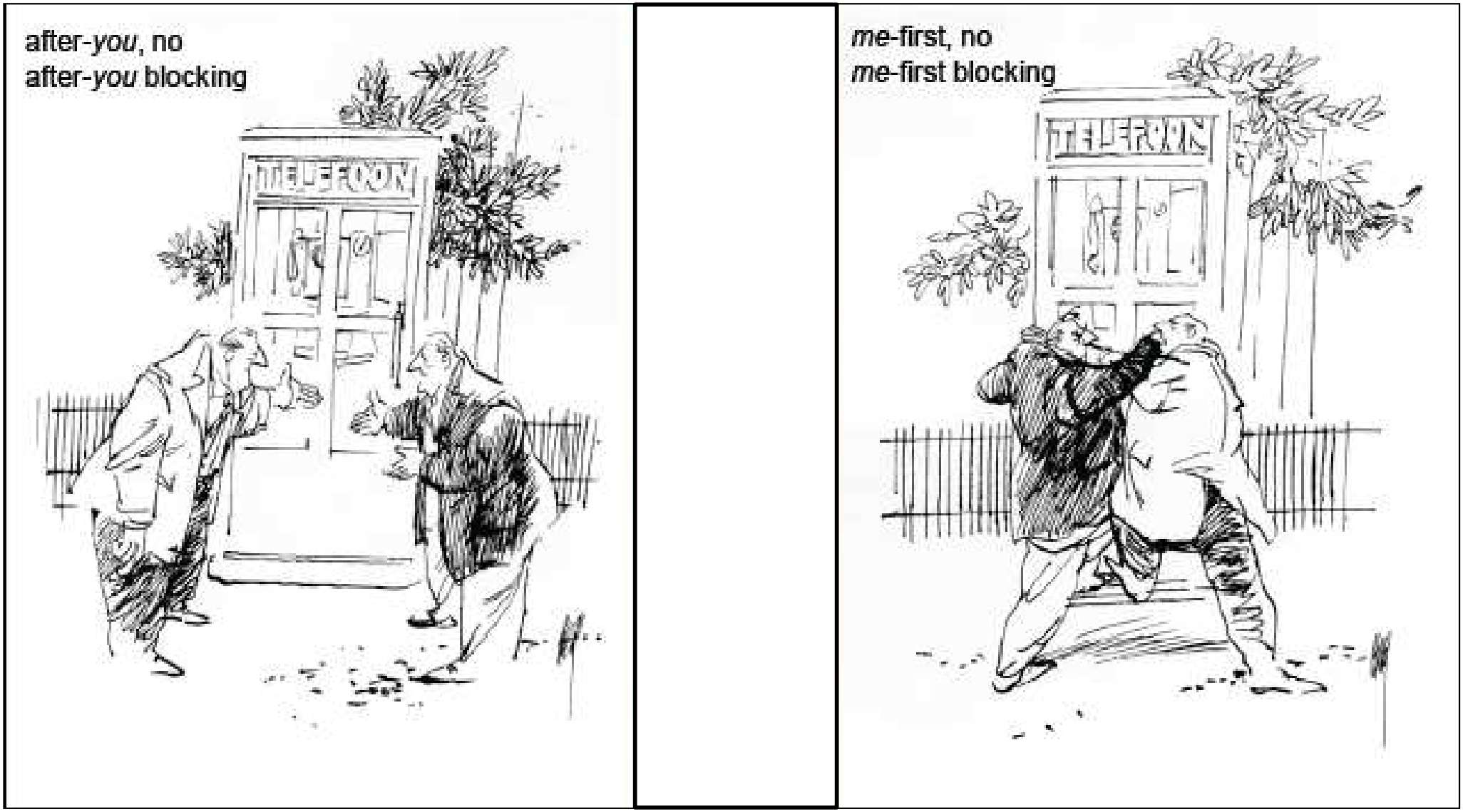
- Regola di precendenza ad un incrocio

  “i veicoli che procedono da destra hanno sempre la precedenza”
- Problema: tutte le direzioni hanno un veicolo in avvicinamento (deadlock)
- Dare la precedenza a sinistra non risolve il problema
- Soluzioni possibili:
  - I veicoli all’interno dell’incrocio hanno la precedenza
  - Uso di semafori



- Non sempre queste soluzioni riescono a evitare ingorghi
  - Intervento umano in genere risolve queste situazioni

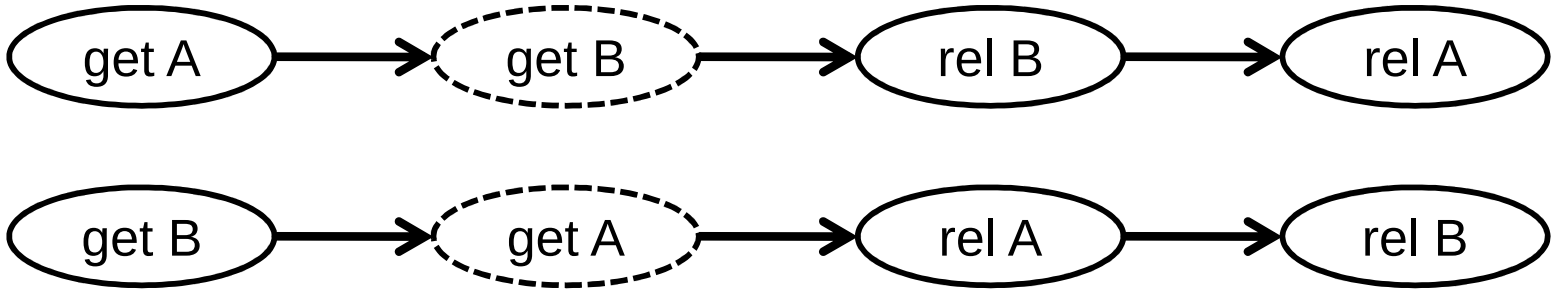
Algoritmi con una regola fissa



“Abbraccio mortale”

- Pippo vuole telefonare a Paperino e Paperino vuole telefonare a Pippo
- Prendono il ricevitore contemporaneamente e non lo lasciano finché non riescono a chiamare (deadlock)

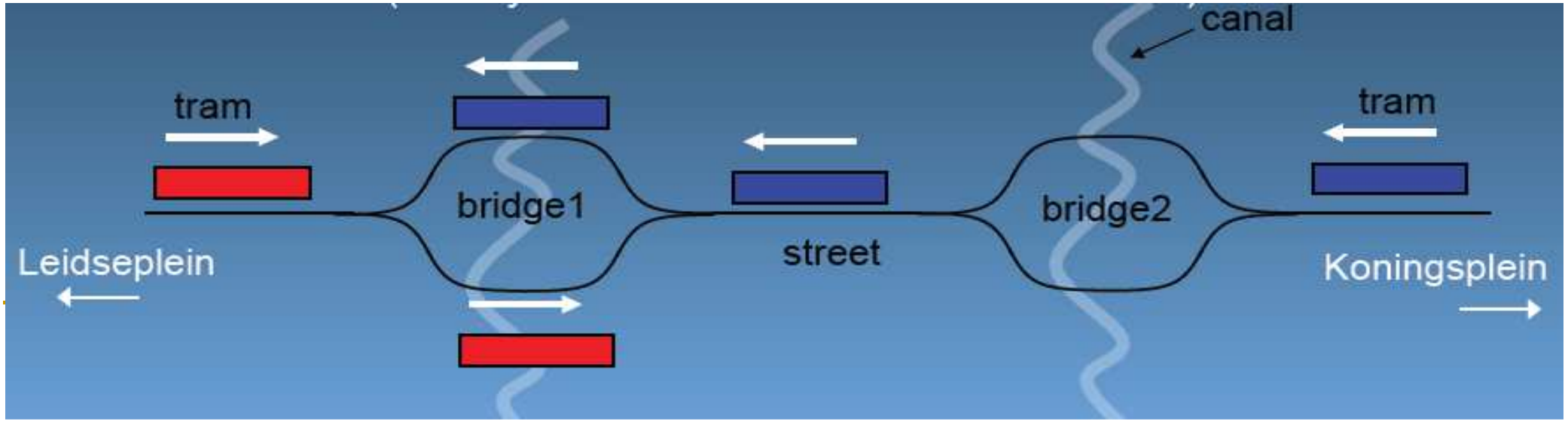
- Situazioni analoghe accadono nei sistemi operativi quando si devono richiedere due risorse per eseguire un lavoro



Problema con risorse condivise in pratica

Un semplice problema di traffico con risorse condivise

- Regola di priorità: tram verso il centro
- Sfida: realizzare un sistema di semafori per evitare deadlock e starvation (attesa illimitata) massimizzando l'attraversamento dei tram



Realizzare sistemi corretti è difficile

- Regola condivisa di system engineering:

  “I sistemi complessi devono essere costruiti da componenti semplici, ciascuna progettata e testata con alta affidabilità”

- Alcuni problemi tuttavia sono visibili solo a livello di sistema

- Testare tutti i comportamenti di interi sistemi può essere semplicemente non fattibile
  - i modi di interagire delle componenti sono elevati

Incidente aereo in Polonia 14/9/1993

- Un aereo Lufthansa Airbus 320-200 con 72 persone a bordo è uscito fuori pista all'aeroporto di Varsavia in fase di atterraggio durante un nubifragio (2 morti)
- Cause dell'incidente:
  - l'inversione della spinta dei reattori si è attivata in ritardo
- Le singole componenti dell'Airbus non hanno avuto malfunzionamenti e i piloti hanno seguito la procedura correttamente

Incidente aereo in Polonia 14/9/1993

- Il ritardo è stato causato dal fatto che l'inversione della propulsione viene attivata solo quando l'aereo è a terra
- Per l'effetto dell'acqua-planing il sistema di controllo ha realizzato che l'aereo aveva toccato terra con 9 secondi di ritardo
- Per scoprire questo errore di progettazione nel sistema di controllo occorreva considerare una serie di coincidenze improbabili, difficilmente immaginabili da un essere umano

Cosa ci insegnano questi esempi?

- La concorrenza è una caratteristica di molti sistemi della vita reale
- I problemi collegati alla concorrenza non sono casi singolari che si manifestano in remoti angoli dell'ingegneria del software
- E' tutt'altro che semplice comprendere e prevedere i comportamenti di un sistema concorrente
  - Anche delle regole ovvie possono avere conseguenze inattese

Individuazione di errori

- Non basta analizzare le singole componenti per scoprire i malfunzionamenti
- Data la complessità, disporre di strumenti automatici è essenziale
- Il testing in genere non riesce ad individuare alcuni problemi tipici dei sistemi concorrenti
  - limitata osservabilità (testing esaustivo non possibile)
  - limitata riproducibilità (due esecuzioni di uno stesso test case possono produrre comportamenti differenti)

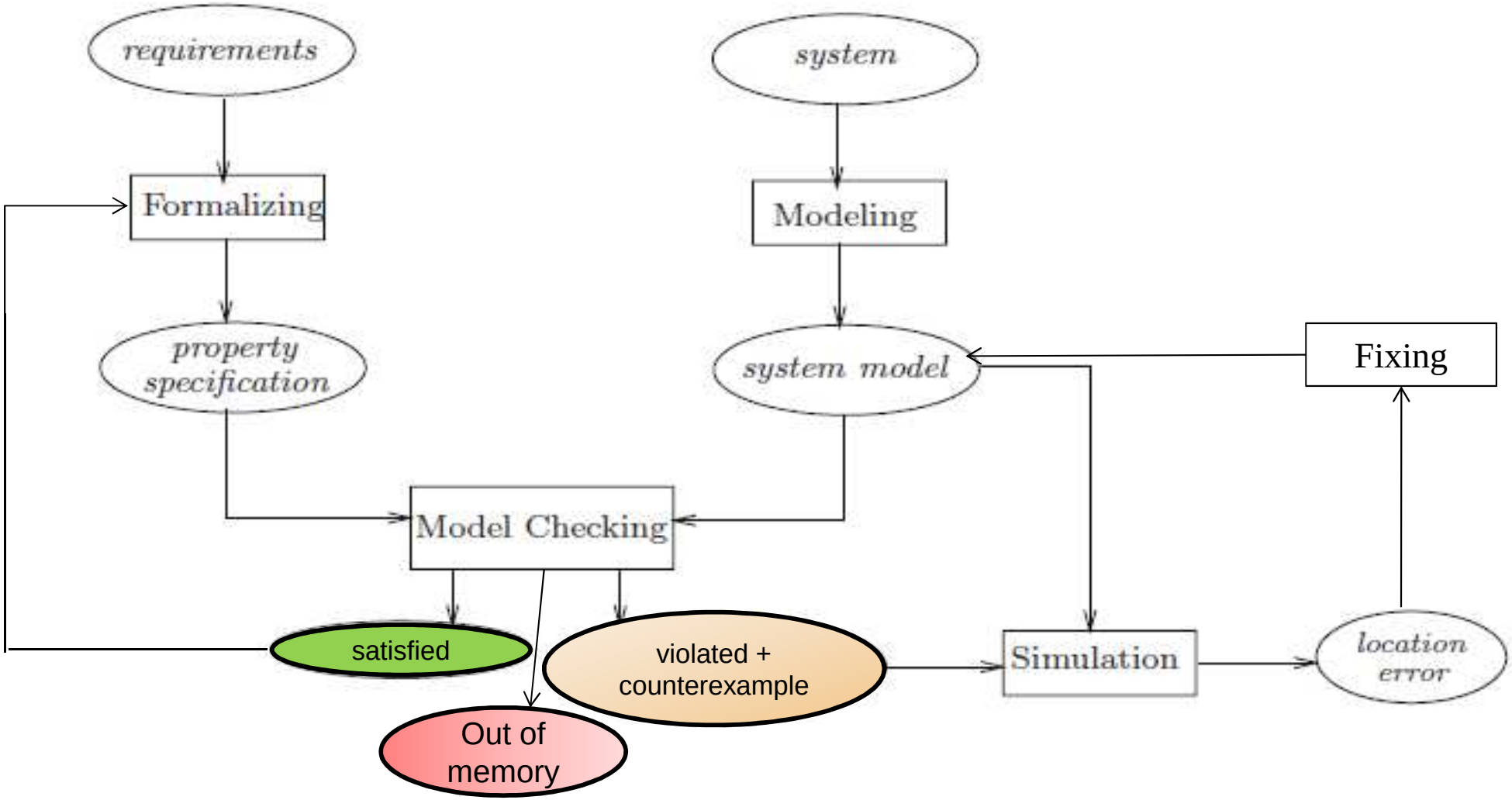
Verifica automatica formale

- Occorre un metodo diverso di verifica che permetta di prendere in considerazione tutti i possibili comportamenti

- Model-checking:
  - si considera un modello astratto del sistema e una specifica formale
  - si verifica che il modello rispetta la specifica

- Efficace nel rilevare gli errori di progettazione a livello di sistema (e non solo a livello di componente)

Model-checking loop



Model-checking process

- *Modeling phase*
  - modella il sistema
  - esegui alcune simulazioni per verificare accuratezza modello
  - formalizza la proprietà da verificare
- *Running phase*
  - esegui il model-checker per verificare la validità della proprietà nel modello
- *Analysis phase*
  - Proprietà soddisfatta? → *testa la prossima proprietà (se esiste)*
  - Proprietà violata? →

    1. analizza il controesempio generato utilizzando la simulazione
    2. raffina il modello, progetto, o proprietà *. . . e ripeti la procedura*
  - out of memory? → *prova a ridurre il modello e riprova*

Punti di forza del model-checking

- *approccio di verifica generale*
- *supporta verifica parziale (le proprietà possono essere verificate individualmente)*
- *indipendente dalla probabilità che un errore accada*
- *genera controesempi*
- *“push-button” technology*
  - *l’uso del model-checking non richiede nè un elevata interazione con l’utente nè un elevato grado di esperienza*
- *crescente interesse da parte dei produttori di sistemi digitali*
- *può essere facilmente integrato nei cicli di progettazione e sviluppo esistenti*
- *può accorciare sensibilmente i tempi di realizzo dei sistemi*
- *basato su una solida teoria*
  - *logica, automi, algoritmi su grafi, strutture dati.*

Punti deboli del model checking

- *appropriato per sistemi control-intensive*
- *meno adatto per sistemi data-intensive* (dove i dati sono su domini infiniti)
- *problemi di decidibilità e complessità (state-space explosion problem)*
- *verifica un modello e non il sistema vero e proprio*
  - qualità dell'analisi dipendente dalla qualità del modello
- *verifica solo le proprietà enunciate*
- *uso richiede un po' di esperienza nel trovare le astrazioni appropriate e nell'enunciare le proprietà nel formalismo logico utilizzato*
- *un model checker è un programma*
  - può contenere errori