Cenni a tecniche di analisi

Analisi Statica e Dinamica dei programmi

Tecniche di verifica – 1/2

- Testing
  - Volto a rilevare malfunzionamenti esercitando il software sotto test con input ed osservando gli output
- Analisi statica
  - prende in esame le istruzioni del programma (che sono un insieme finito), senza eseguirlo
  - è impossibile rilevare malfunzionamenti che dipendono dal valore assunto dinamicamente dalle variabili
  - è più facile rilevare anomalie legate alla presenza di variabili non dichiarate in linguaggi fortemente tipizzati quali Pascal, Ada, C++, etc.

Tecniche di verifica - 2/2

- Analisi dinamica
  - prende in esame l'insieme delle esecuzioni di un programma (quasi mai limitato)
  - vi è il problema di ridurre il numero di casi da esaminare

- Le varie tecniche sono spesso usate in maniera complementare

Analisi

- Le tecniche di analisi includono
  - Tecniche di ispezione manuale
  - Analisi automatica

- **L'analisi mira a verificare il soddisfacimento di una proprietà**
- Può essere applicata ad ogni fase del processo di sviluppo
- È particolarmente idonea nelle prime fasi, nella specifica e nella progettazione

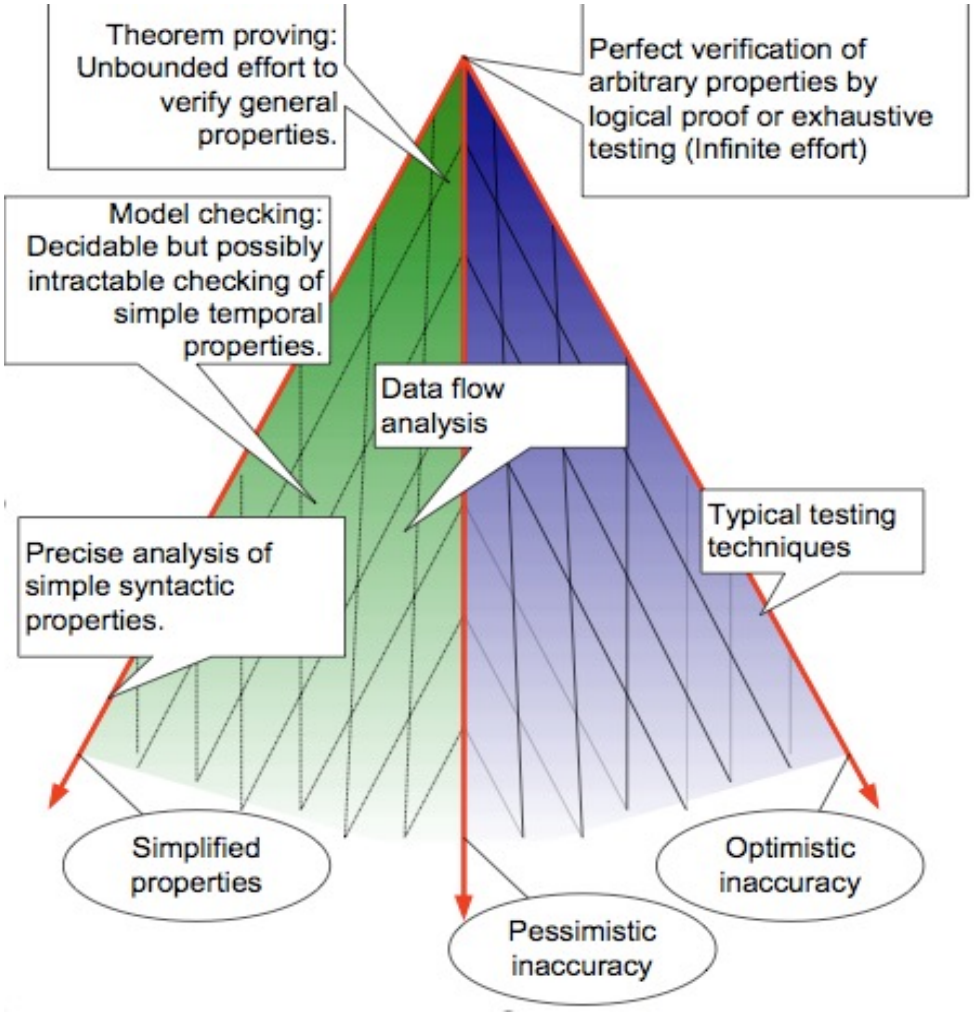
Ispezione

- Può essere applicata ad ogni documento
  - Requisiti
  - Documenti di progetto
  - Piani di test e casi di test
  - Codice sorgente
- Richiede una notevole quantità di tempo
  - Re-ispezionare un componente che è cambiato può essere molto costoso
- Usata principalmente
  - Quando le altre tecniche sono inapplicabili
  - Quando le altre tecniche non forniscono una sufficiente copertura

Analisi statica

- Di applicabilità più limitata
  - Può essere applicata a rappresentazioni formali dei requisiti
  - Non a documenti espressi in linguaggio naturale

- Quando può essere applicata, è particolarmente efficiente, perchè consente di sostituire cicli macchina all'intervento umano per attività molto ripetitive e error-prone



Punti di vista sulle tecniche

- Inaccuratezza Ottimistica: si accettano programmi che non posseggono la proprietà analizzata (i.e., possiamo non rilevare tutte le violazioni)
  - Testing
- Inaccuratezza pessimistica: non è garantita l'accettazione del programma anche se possiede la proprietà analizzata
  - Tecniche di analisi automatica
- Proprietà semplificate: riduce il grado di libertà per semplificare la proprietà da controllare

Why Analysis

- **Exhaustively check properties that are difficult to test**
  - Faults that cause failures
    - rarely
    - under conditions difficult to control
  - Examples
    - race conditions
    - faulty memory accesses
- Extract and summarize information for inspection and test design

Why automated analysis

- Manual program inspection
  - effective in finding faults difficult to detect with testing
  - But humans are not good at
    - repetitive and tedious tasks
    - maintaining large amounts of detail
- Automated analysis
  - replace human inspection for some class of faults
  - support inspection by
    - automating extracting and summarizing information
    - navigating through relevant information

Static vs dynamic analysis

- Static analysis
  - examine program source code
    - examine the complete execution space
    - but may lead to false alarms
- Dynamic analysis
  - examine program execution traces
    - no infeasible path problem
    - but cannot examine the execution space exhaustively

Concurrency faults

- Concurrency faults
  - deadlocks: threads blocked waiting each other on a lock
  - data races: concurrent access to modify shared resources
- Difficult to reveal and reproduce
  - nondeterministic nature does not guarantee repeatibility
- Prevention
  - Programming styles
    - eliminate concurrency faults by restricting program constructs
    - examples
      - do not allow more than one thread to write to a shared item
      - provide programming constructs that enable simple static checks (e.g., Java synchronized)
- Some constructs are difficult to check statically
    - example
      - C and C++ libraries that implement locks

Memory faults

- Dynamic memory access and allocation faults
  - null pointer dereference
  - illegal access
  - memory leaks
- Common faults
  - buffer overflow in C programs
  - access through *dangling* pointers
  - slow leakage of memory
- Faults difficult to reveal through testing
  - no immediate or certain failure

Example

```
} else if (c == '%') {
   int  digit_high = Hex_Values[*(++eptr)];
   int  digit_low  = Hex_Values[*(++eptr)];
```

- fault
  - input string terminated by an hexadecimal digit
  - scan beyond the end of the input string and corrupt memory
  - failure may occur much after the execution of the faulty statement
- hard to detect
  - memory corruption may occur rarely
  - lead to failure more rarely

Memory Access Failures

- (explicit deallocation of memory - C,C++)
- Dangling pointers: deallocating memory accessible through pointers
- Memory leak: failing to deallocate memory not accessible any more
  - no immediate failure
  - may lead to memory exhaustion after long periods of execution
    - escape unit testing
    - show up only in integration, system test, actual use
- can be prevented by using
  - program constructs
    - saferC (dialect of C used in avionics applications) limited use of dynamic memory allocation -> eliminates dangling pointers and memory leaks (restriction principle)
  - analysis tools
    - Java dynamic checks for out-of-bounds indexing
    - Automatic storage deallocation (garbage collection)

FLOW ANALYSIS

Control Flow Analysis

- Il flusso di controllo è esaminato per verificarne la correttezza
- Il codice è rappresentato tramite un grafo, il grafo del flusso di controllo (Control flow Graph - CfG), i cui nodi rappresentano statement (istruzioni eo predicati) del programma e gli archi il passaggio del flusso di controllo.
- Il grafo è esaminato per identificare ramificazioni del flusso di controllo e verificare l'esistenza di eventuali anomalie quali codice irraggiungibile e non strutturazione.

Data Flow Analysis

- Analizza l'evoluzione del valore delle variabili durante l'esecuzione di un programma, permettendo di rilevare anomalie.
- L'analisi statica è legata alle operazioni eseguite su una variabile:
  - **Definizione:** alla variabile è assegnato un valore
  - **Uso:** il valore della variabile è usato in un'espressione o unpredicato
  - **Annullamento:** al termine di un'istruzione il valore associato alla variabile non è più significativo
  - *Es. nell'espressione a:=b+c la variabile a è definita mentre b e c sono usate*
- *Dalle sequenze osservate, è possiblie eseguire diverse analisi (es. Live, Avail, Reaching Def)*

Data Flow Analysis

- La definizione di una variabile , così come un annullamento, cancella l'effetto di una precedente definizione della stessa variabile , ovvero ad essa è associato il nuovo valore derivante dalla nuova definizione (o il valore nullo)
- Una corretta sequenza di operazioni prevede che:
  - L'uso di una variabile x deve essere sempre preceduto da una definizione della stessa variabile x , senza annullamenti intermedi
  - Un uso non preceduto da una definizione può corrispondere al potenziale uso di un valore non determinato

Data Flow Analysis

- Una definizione di una variabile x deve essere sempre seguita da un uso della variabile x, prima di un'altra definizione o di un annullamento della stessa variabile x
  - Una definizione non seguita da un uso corrisponde all'assegnamento di un valore non utilizzato e quindi potenzialmente inutile

Data Flow Analysis

- Sequenze di istruzioni sono riconducibili a sequenze di definizioni (d), usi (u), annullamenti (a) delle variabili referenziate nei comandi

```
Procedure swap (x1, x2: real)
var x:real
begin
   x2:=x;
   x2:=x1;
   x1:=x;
end;
```

x: (auu)

x2: (ddd)

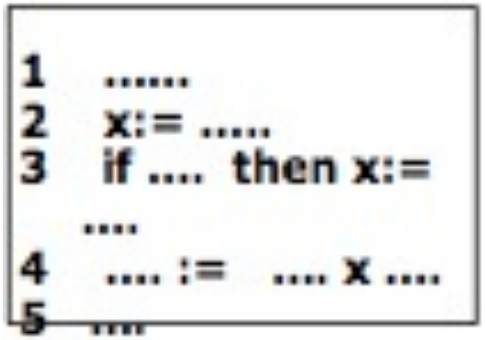
x1: (dud)

La sequenza (auu) di x e la sequenza (ddd) di x2 sono indicative di una qualche anomalia

Data Flow Analysis

- Non sempre le sequenze ...***au***... o ...***dd***... corrispondono ad anomalie
- Es.
- ***au*** può comparire in un generatore di numeri casuali (è letto il contenuto di una cella di memoria non inizializzata per determinare il seme della generazione)
- ***dd*** può dipendere da una cattiva strutturazione del programma (la prima definizione non è usata nella esecuzione considerata ma lo è un un'altra, su un altro cammino)

Data Flow Analysis



Nelle linee 2 e 3 due definizioni staticamente consecutive di x; la definizione della linea 2 è usata nella linea 4 che in alcune esecuzioni può seguire direttamente la linea 2

- Sequenze di azioni su una variabile per una determinata esecuzione (cammino) sono rappresentabili tramite espressioni

Data Flow Analysis

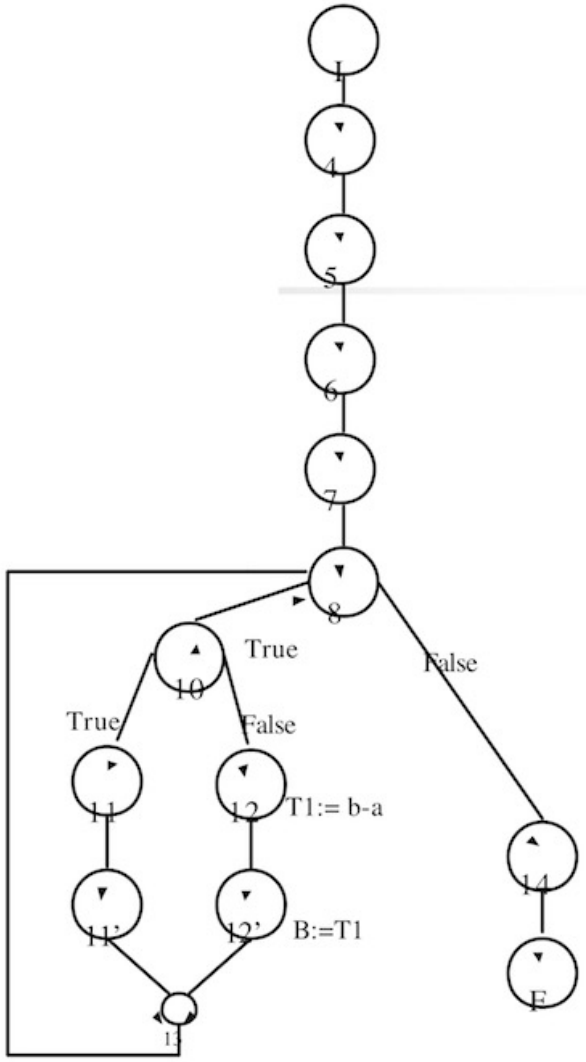
- L'espressione relativa ad un cammino p di un programma P per la variabile x è indicata con P(p;x)

```
1   ……
2   x:= …..
3   if …. then x:= ….
4   …. :=  ….x ….
5   ….
```

- P([..., 2, 4, ...]; *x) = (...du...)*
- *P([..., 2, 3, 4, ...]; x) = (...ddu...)*

Example

- 1 .program gcd (input, output);
- 2. var x,y,a,b : integer;
- 3. Begin
- 4. read (x,y);
- 5. a:=x;
- 6. a:=y
- 7. while a <> b do
- 8. Begin
- 9. if a>b
- 10. then a:=a-b
- 11. else b:=b-a;
- 12. end;
- 13. write (“il massimo comune divisore è”, a)
- 14. end.



Data Flow Analysis

- A ciascun nodo del CfG è possibile associare l'insieme delle variabili definite in esso, quello delle variabili usate e quello delle variabile annullate.

Nodo	Var. definite	Var usate	Var. annullate
4			x, y, a, b
5	x, y		
6	a	x	
7	a	y	
8		a, b	
10		a, b	
11		a, b	
11'	a		
12		a, b	
12'	b		
14		a	

$$P([I,4,5,6,7,8,10,11,11',8,14,F];a) = (\text{-a-dduuuduu-})$$

$$P([I,4,5,6,7,8,10,11,11',8,14,F];b) = (\text{-a---uuu-u--})$$

E' quindi possibile scrivere l'espressione P(p;x) facendo riferimento a tali insiemi

Anomalia **...dd... per a** Anomalia **...a---u... per b** dovuta ad errore a linea 7.
7 a=y invece di b:=y;

SYMBOLIC EXECUTION

Symbolic Execution

- Il programma non è eseguito con i valori effettivi ma con valori simbolici dei dati di input
- L'esecuzione procede come una esecuzione normale ma non sono elaborati valori bensì formule formate dai valori simbolici degli input
- Gli output sono formule dei valori simbolici degli input
- Si basa su ***precondizioni, invarianti e post-condizioni, sottoforma di asserzioni.*** Tramite constraint solver, è valutata la correttezza dell'algoritmo nel soddisfare la post-condizione date le precondizioni

Symbolic Execution

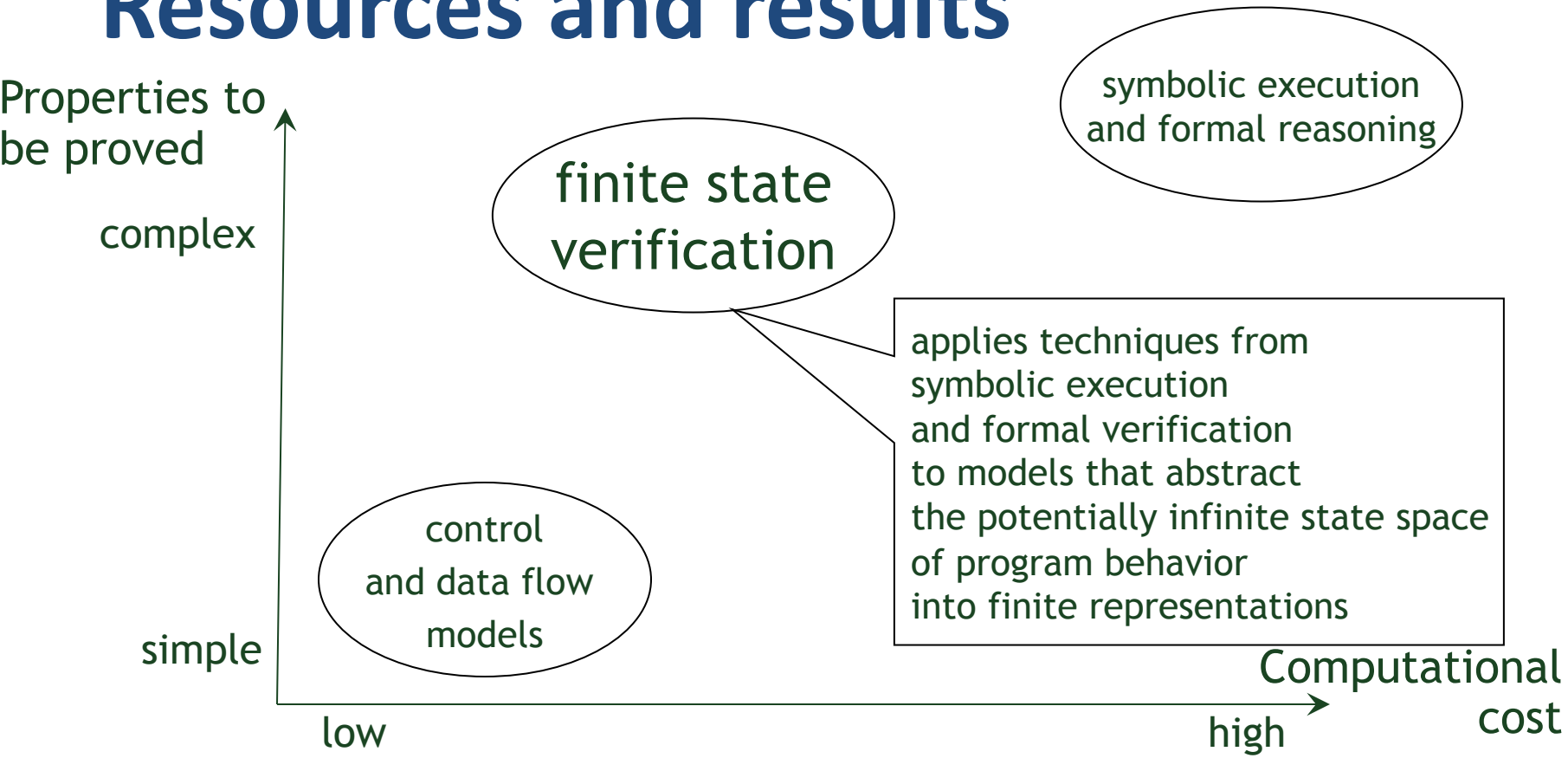
- Può risultare molto onerosa: istruzioni condizionali (deve essere valutato ciascun caso (vero e falso)) e programmi con cicli.
- Trovare e verificare un insieme completo di asserzioni è complicato
- Questa tecnica è perciò riservata per moduli piccoli e estremamente critici.

FINITE STATE VERIFICATION

Limits and trade-offs

- Finite state verification can automatically prove *some* significant properties of a *finite model* of the infinite execution space
  - balance trade-offs among
    - generality of properties to be checked
    - class of programs or models that can be checked
    - computational effort in checking
    - human effort in producing models and specifying properties

Resources and results



Cost trade-offs

- Human effort and skill are required
  - to prepare a finite state model
  - to prepare a suitable specification for automated analysis
- Iterative process:
  - prepare a model and specify properties
  - attempt verification
  - receive reports of impossible or unimportant faults
  - refine the specification or the model
- Automated step
  - computationally costly
    - computational cost impacts the cost of preparing model and specification, which must be tuned to make verification feasible
  - manually refining model and specification less expensive with near-interactive analysis tools

Applications for Finite State Verification

- Concurrent (multi-threaded, distributed, ...)
  - Difficult to test thoroughly (apparent non-determinism based on scheduler); sensitive to differences between development environment and field environment
  - First and most well-developed application of FSV
- Data models
  - Difficult to identify “corner cases” and interactions among constraints, or to thoroughly test them
- Security
  - Some threats depend on unusual (and untested) use

Pointer Analysis

- Pointer variable represented by a machine with three states:
  - invalid value
  - possibly null value
  - definitely not null value
- Deallocation triggers transition from non-null to invalid
- Conditional branches may trigger transitions
  - E.g., testing a pointer for non-null triggers a transition from possibly null to definitely non-null
- Potential misuse
  - Deallocation in possibly null state
  - Dereference in possibly null
  - Dereference in invalid states

Merging States

- Flow analysis
  merge states obtained along different execution paths
  - conventional data flow analysis: merge all states encountered at a particular program location
  - FSM: summarize states reachable along all paths with a set of states
- Finite state verification techniques
  never merge states (path sensitive)
  - procedure call and return:
    - complete path- and context-sensitive analysis → too expensive
    - throwing away all context information → too many false alarms
    - symbolic testing: cache and reuse (entry, exit) state pairs

DYNAMIC ANALYSIS

Buffer Overflow

```
…
int main (int argc, char *argv[]) {
     char sentinel_pre[] = "2B2B2B2B2B";
     char subject[]  = "AndPlus+%26%2B+%0D%";
    char sentinel_post[] = "26262626";
    char *outbuf = (char *) malloc(10);
    int return_code;

     printf("First test, subject into outbuf\n");
    return_code = cgi_decode(subject, outbuf);
    printf("Original: %s\n", subject);
    printf("Decoded:  %s\n", outbuf);
    printf("Return code: %d\n", return_code);

     printf("Second test, argv[1] into outbuf\n");
     printf("Argc is %d\n", argc);
    assert(argc == 2);
    return_code = cgi_decode(argv[1], outbuf);
    printf("Original: %s\n", argv[1]);
    printf("Decoded:  %s\n", outbuf);
    printf("Return code: %d\n", return_code);
}…
```

Output parameter of fixed length
Can overrun the output buffer

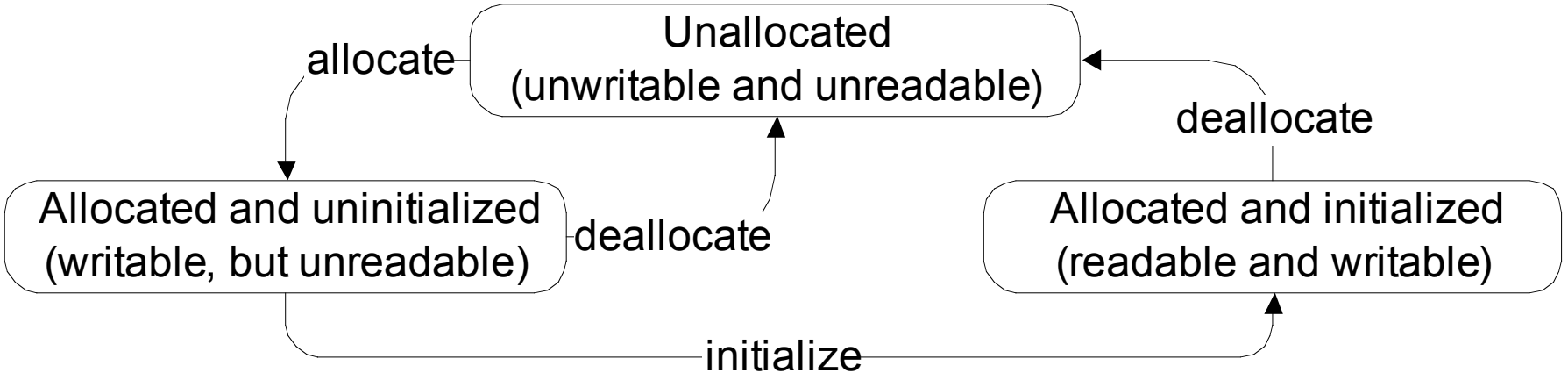
Dynamic Memory Analysis (with Purify)

```
[I] Starting main
[E] ABR: Array bounds read in printf {1 occurrence}
    Reading 11 bytes from 0x00e74af8 (1 byte at 0x00e74b02 illegal)
    Address 0x00e74af8 is at the beginning of a 10 byte block
    Address 0x00e74af8 points to a malloc'd block in heap 0x00e70000
    Thread ID: 0xd64
...
[E] ABR: Array bounds read in printf {1 occurrence}
    Reading 11 bytes from 0x00e74af8 (1 byte at 0x00e74b02 illegal)
    Address 0x00e74af8 is at the beginning of a 10 byte block
    Address 0x00e74af8 points to a malloc'd block in heap 0x00e70000
    Thread ID: 0xd64
...
[E] ABWL: Late detect array bounds write {1 occurrence}
    Memory corruption detected, 14 bytes at 0x00e74b02
    Address 0x00e74b02 is 1 byte past the end of a 10 byte block at 0x00e74af8
    Address 0x00e74b02 points to a malloc'd block in heap 0x00e70000
    63 memory operations and 3 seconds since last-known good heap state
    Detection location - error occurred before the following function call
     printf          [MSVCRT.dll]
...
        Allocation location
     malloc          [MSVCRT.dll]
...
[I] Summary of all memory leaks... {482 bytes, 5 blocks}
...
[I] Exiting with code 0 (0x00000000)
    Process time: 50 milliseconds
[I] Program terminated ...
```

Identifies the problem

Memory Analysis

- Instrument program to trace memory access
  - record the state of each memory location
  - detect accesses incompatible with the current state
    - attempts to access unallocated memory
    - read from uninitialized memory locations
  - array bounds violations:
    - add memory locations with state *unallocated* before and after each array
    - attempts to access these locations are detected immediately



Data Races

- Testing: not effective
  (nondeterministic interleaving of threads)
- Static analysis:
  computationally expensive, and approximated
- Dynamic analysis:
  can amplify sensitivity of testing to detect potential data races
  - avoid pessimistic inaccuracy of finite state verification
  - Reduce optimistic inaccuracy of testing

Dynamic Lockset Analysis

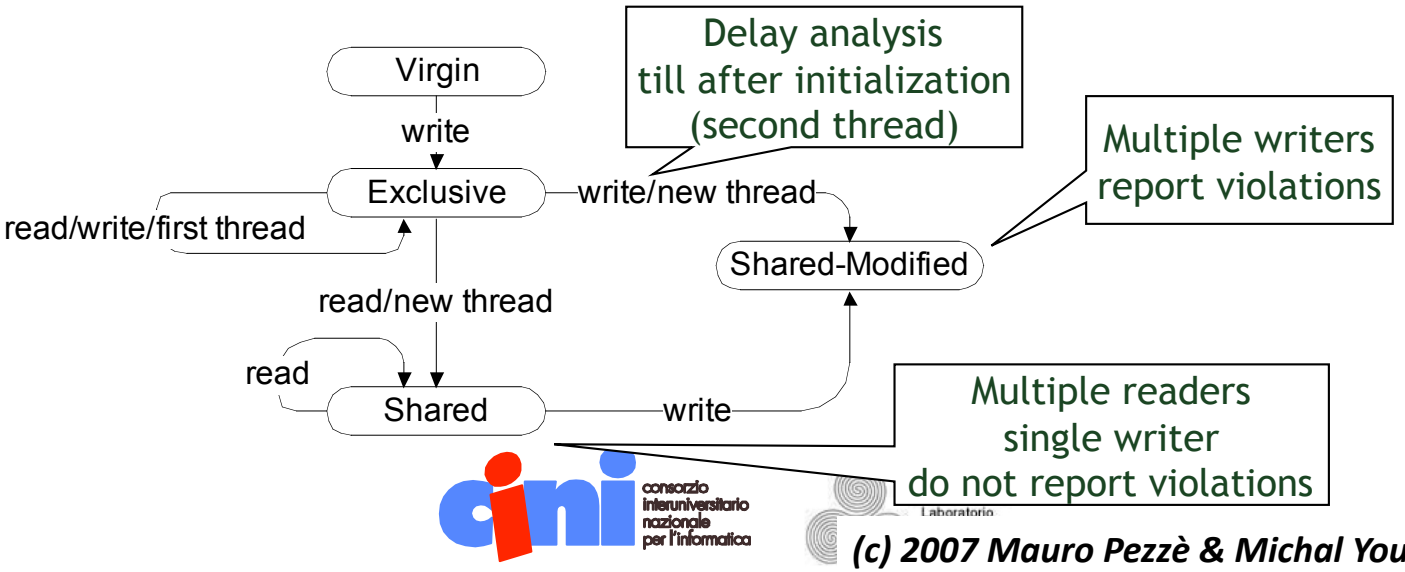
- Lockset discipline: set of rules to prevent data races
  - Every variable shared between threads must be protected by a mutual exclusion lock
  - ….
- Dynamic lockset analysis detects violation of the locking discipline
  - Identify set of mutual exclusion locks held by threads when accessing each shared variable
  - INIT: each shared variable is associated with all available locks
  - RUN: thread accesses a shared variable
    - intersect current set of candidate locks with locks held by the thread
  - END: set of locks after executing a test = set of locks always held by threads accessing that variable
    - empty set for v = no lock consistently protects v

Simple lockset analysis: example

Thread	Program trace	Locks held	Lockset(x)	
		{}	{lck1, lck2}	INIT:all locks for x
thread A	lock(lck1)			
		{lck1}		lck1 held
	x=x+1			
			{lck1}	Intersect with locks held
	unlock(lck1}			
		{}		
tread B	lock{lck2}			
		{lck2}		
	x=x+1			lck2 held
			{}	
	unlock(lck2}			Empty intersection potential race
		{}		

Handling Realistic Cases

- simple locking discipline violated by
  - initialization of shared variables without holding a lock
  - shared variables writing only during initialization without locks
  - allowing multiple readers in mutual exclusion with single writers



Using Behavioral Models

- Testing
  - validate tests thoroughness
- Program analysis
  - understand program behavior
- Regression testing
  - compare versions or configurations
- Testing of component-based software
  - compare components in different contexts
- Debugging
  - Identify anomalous behaviors and understand causes