Motivation for Software Verification

- Last year, at 12.30h, I got the following dialog box:

  The application PowerPoint quit unexpectedly.
  Mac OS X and other applications are not affected.
  Click Reopen to open the application again. Click Report to see more details or send a report to Apple.

  Close  Report...  Reopen

- As usual, PowerPoint’s "AutoRecovery File" did not work…
- Microsoft: please start using “Software Verification” technology!
**Classic vs Modern Approach**

**Classic MC**
- (initial) Design
- (manual) abstractions
- Abstract Verification Model
  - refinement techniques
- Implementation

**Software MC**
- Abstract Verification Model
  - abstraction techniques
- Implementation

---

**Model Checking Programs (1)**

- **Formal Methods** research (> 30 years) traditionally has been devoted to (small) **special languages**:
  - formal specification languages (e.g. LOTOS, Z, B)
  - logic based languages in theorem proving (e.g. HOL, PVS)
  - guarded command languages (e.g. Murphi, SMV, Uppaal)
  - resembling programming languages (e.g. Promela)

- **Research on software model checking is young**
  - 10 years ago: no research / tools available
  - 5 years ago: just a few, with limited functionality
  - now: Bandera, BLAST, Bogor, dSPIN, ESC/Java, FeaVer, JPF, SLAM, SPIN 5, VeriSoft, MoonWalker, etc.
  
  *not longer "toy examples", but (towards) industrial systems*

---

**Model Checking Programs (2)**

- **FM**: verification technology should be applied to designs.
  - Catching errors as early as possible is crucial.

- Compelling reasons to also try to verify programs:
  - Programs often contain fatal errors (despite a careful design).
    - e.g. concurrency errors introduced in the code
  - Modern programming languages are the result of good language design principles.
    - distinction between design and program gets blurred (UML)
  - Force the FM community to deal with very hard problems.
    - Furthermore, the objective of formal methods is not only to prove programs correct, but also to debug programs.
  - Make it feasible to compare and integrate different tools working on the same standardised language.

**Goal**: formal methods should play a role for everyday software developers.

---

**Software Model Checking (1)**

- **Translation** based
  - An existing model checker is (mis)used for the verification of the source code.
    - Java PathFinder v1
      - [Havelund & Pressburger 2000]
    - FeaVer / Modex
      - [Holzmann & Smith 2000]

The counter example has to be interpreted in terms of the original source program.
**Software Model Checking (1)**

- **Translation** based
  - An existing model checker is (mis)used for the verification of the source code.
  - Java PathFinder v1
    [Havelund & Pressburger 2000]
  - FeaVer / Modex
    [Holzmann & Smith 2000]

**Software Model Checking (2)**

- **Intermediate Representation** based
  - The program is translated to an intermediate representation which is then analyzed using a virtual machine.
  - JPF
    [Visser et al. 2003]
    http://javapathfinder.sourceforge.net/
  - XRT
    [Grieskamp et al. 2005]
  - BOGOR framework
    [Robby et al. 2003]
  - MoonWalker
    http://www.cs.utwente.nl/~ruys/moonwalker/

**Software Model Checking (3)**

- **Abstraction** based
  - An abstract, overapproximated model is analyzed using a model checker.
  - SLAM
    [Ball & Rajamani 2002]
    http://javapathfinder.sourceforge.net/
  - BLAST
    [Menzinger, Jhala & Majumdar 2005]

This type of model checking is now called CESAR (= counter example guided abstraction refinement).

---

**Overview**

- **Formal Methods** (i.e. model checking) meets **Compiler Construction** (i.e. static analysis).

**Gerard J. Holzmann.**
**From Code to Models.**

Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoong Park and Flavio Lerda.
Model Checking Programs.

Thomas Ball and Sriram K. Rajamani.
Automatically Validating Temporal Safety Properties of Interfaces.

---

**Software Model Checking (4)**

- **Translation** based
  - An existing model checker is (mis)used for the verification of the source code.
  - Java PathFinder v1
    [Havelund & Pressburger 2000]
  - FeaVer / Modex
    [Holzmann & Smith 2000]

The counter example has to be interpreted in terms of the original source program.
**Modex/SPIN (1)**

- **FeaVer** [Holzmann & Smith 2000]
  - First case where model checking was used to analyse a large application in a commercial setting.
    - Lucent's PathStar access server (a POTS)
    - 2 million lines of C code
    - 50 features (call waiting, call forwarding, etc.)
    - 2^50 possible feature interactions
  - model checker found an order of magnitude more errors than the traditional testing team (75 vs 5).
- Semi-automatic model extraction from C programs
  - manual creation of a translation table (C ↔ Promela)
  - extraction itself is then automatic
- Works well when source code is relatively stable.

---

**Modex/SPIN (2)**

FeaVer Test Harness

- C program
- Counterexamples
- Table to translate: C ↔ Promela
- Promela code to "drive" the code. E.g. to simulate the environment.
- LTL property
  - as Promela "never claim".

---

**Modex/SPIN (3)**

- Abstraction / filtering
  - the label set of the extracted automaton represents
    - declarations, statements, and expressions from C
  - each label is classified (with a lookup table):
    - irrelevant to the property hide
    - partially relevant map
    - fully relevant keep
  - a modified C parser (modex) uses the table to generate a SPIN verification model (i.e. a Promela model).
    - in PathStar application: 30% hide, 10% map, 60% keep

  **E.g.**

  ```
  set_timer(160000)  timer!Set(p0,160000)
  reset_timer()      timer!Reset(p0,0)
  send(p0,Msg)       p0!Msg
  resp=wait_recv()   q0?resp
  ```

  Abstraction occurs since very complex lines of C code can be replaced by simple abstract code in Promela.

---

**Overview**

Gerard J. Holzmann.
From Code to Models.

Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park and Flavio Lerda.
Model Checking Programs.

Thomas Ball and Sriram K. Rajamani.
Automatically Validating Temporal Safety Properties of Interfaces.
**Software Model Checking (2)**

- **Intermediate Representation** based
  - The program is translated to an intermediate representation which is then analyzed using a virtual machine.
- **JPF**
  - [Visser et al. 2003](http://javapathfinder.sourceforge.net/)
- **XRT**
  - [Grieskamp et al. 2005](http://www.cs.utsa.edu/~grieskamp/)
- **BOGOR framework**
  - [Robby et al. 2003](http://www.cs.utsa.edu/~grieskamp/)
- **MoonWalker**

---

**Model Checking Java**

- Reasons why model checking programs is considered hard:
  - complex language constructs
    - Input languages for model checkers are often kept relatively simple, to allow efficient processing during model checking.
  - complex states
    - State of a software system is considered too complex to be encoded efficiently.
  - state space explosion
    - State space of a program is often infinitely large.

---

**Java PathFinder (JPF)**

- **Java PathFinder v2 (JPF)** uses custom-made model checker that can execute all Java bytecode instructions.
  - own JVM: JPF supports all Java bytecodes
  - depth-first traversal with backtracking (like SPIN)
  - major design decision: JPF should be modular and understandable (speed is sacrificed) - written in Java
    - SPIN is (at least) one order of magnitude faster
    - JPF v2 developed in 15 man-months
  - JPF can only handle closed systems, i.e. a system and the environment it will execute in.
    - SPIN (and FeaVer) can also only verify closed systems.

---

**Complex States (1)**

- State matching is necessary to ensure termination.
  - one must know when a state is revisited (remember DFS)
- **VeriSoft** considers the state of a software system to be too complex: does not store states.
  - limits the depth of the search to solve termination problem
- A Java state consists of
  - static variables (class variables)
    - including locks
  - dynamic variables (objects)
    - including locks
  - information for each thread
    - stack of frames (one for each method called)
Complex States (2)

- SPIN
  - compressed states are stored in the state space (i.e., hash table)
  - decompression is not needed: original state is stored on the stack

- JPF
  - initially, SPIN-approach
    - compressed states in hash table
    - uncompressed states on stack
  - problem:
    - memory consumption was high for uncompressed states on stack
    - clone() operation is slow for uncompressed state
  - solution:
    - use reverse operation to recreate a state: faster than clone()
    - stack can now only contain the transitions: components that change from one state to the other

Complex States (3)

- Observation on states (revisited):
  - number of distinct system states grows very fast
  - despite the fact that each process and each data object can typically reach only a small number of distinct states
  - explosion of the number of reachable states is caused by the large number of ways in which local states of individual components can be combined.

- Idea of collapse compression
  - store small state components separately
  - assign small unique index numbers to each small component
  - global state descriptor is now formed by the combination of the unique index numbers.

  JPF uses SPIN's collapse compression.

Complex States (4)

- SPIN
  - compressed states are stored in the state space (i.e., hash table)
  - decompression is not needed: original state is stored on the stack

- JPF
  - initially, SPIN-approach
    - compressed states in hash table
    - uncompressed states on stack
  - problem:
    - memory consumption was high for uncompressed states on stack
    - clone() operation is slow for uncompressed state
  - solution:
    - use reverse operation to recreate a state: faster than clone()
    - stack can now only contain the transitions: components that change from one state to the other

Complex States (5)

- JPF
  - initially, SPIN-approach
  - compressed states in hash table
  - uncompressed states on stack
  - problem:
    - memory consumption was high for uncompressed states on stack
    - clone() operation is slow for uncompressed state
  - solution:
    - use reverse operation to recreate a state: faster than clone()
    - stack can now only contain the transitions: components that change from one state to the other

- observation on states (revisited):
  - number of distinct system states grows very fast
  - despite the fact that each process and each data object can typically reach only a small number of distinct states
  - explosion of the number of reachable states is caused by the large number of ways in which local states of individual components can be combined.

- idea of collapse compression
  - store small state components separately
  - assign small unique index numbers to each small component
  - global state descriptor is now formed by the combination of the unique index numbers.

  JPF uses SPIN's collapse compression.
Curbing the State Space Explosion

- Most challenging part!
- Philosophy behind JPF: to integrate techniques from different disciplines:
  - abstraction to turn infinite program into finite one
  - runtime analysis to pinpoint problematic code
  - slicing to reduce the program
  - use model checker to analyse the resulting program
- JPF contains
  - bytecode model checker
  - algorithms to deal with large or infinite state spaces: (i) static analysis, (ii) predicate abstraction, (iii) runtime analysis
  - a custom made JVM^{JPF} which is used for both model checking and runtime analysis

JPF - topics in this lecture

- Introduction
- Symmetry Reductions
- Abstractions
  - over- and under approximations
  - data type abstractions
  - predicate abstractions
- Static Analysis
- Runtime Analysis
  - data race detection
  - deadlock detection

Architecture of JPF

http://javapathfinder.sourceforge.net/

Symmetry Reductions (1)

- Symmetries induce an equivalence relation on states.
  - one can discard a state if an equivalent state has already been explored.
  - canonicalization function maps each state into a unique representative (of the equivalence class)
  - e.g. partial order reduction (equivalence relation on paths)
- Software programs can induce a great many symmetries:
  - class loading: order is not important
  - symmetries in the heap
    - creation of objects by different threads
    - garbage collection (JPF uses mark-and-sweep)

Ensure that the static and dynamic area have a canonical representation regardless of which interleaving of transitions is being executed.
### Symmetry Reductions (2)

- **Static area** (class loading)
  - **First path**: order the classes as they are loaded
    - e.g. Aap → 0, Noot → 1, Mies → 2
  - **During backtracking**: remember the order of the classes
    - e.g. if on some other path, Mies is the first to be loaded, then still assign Mies → 2

- **Dynamic area** (creation of objects on the heap)
  - cannot use the above method because each class can have many objects instantiated
  - uniquely identify each “new” bytecode in the Java program (e.g. the line number in the program)
    - sequence numbers for loops

### Symmetry Reductions (3)

**Example:**

```java
class S { int x; }
class Task1 extends Thread {
    public void run() {
        S s1;
        s1 = new S();
        L1: s1 = new S();
    }
}
class Task2 extends Thread {
    public void run() {
        S s2;
        L2: s2 = new S();
    }
}
```

- **Identify objects by their place in the bytecode program (here symbolically L1 and L2).**
- **First path assigns positions of the objects, backtracking paths remember original positions.**

### Abstraction (1)

- **Three types of abstraction:**
  - **Over-approximations**: more behaviour is added to the abstracted system than is present in the original
  - **Under-approximations**: less behaviour is present in the abstracted system than is present in the original
  - **Precise approximations**: the same behaviour is present in the abstracted system and original system

### Abstraction (2)

- **Under-approximation**
  - Remove parts of the program deemed irrelevant to the property being checked.
    - limit integer values to 0..10 instead of all integer values
    - queue size is 3 instead of unbounded
  - Under-approximation was the abstraction of choice in the early days of program model checking.
    - used during the translation from the input language to the model checker’s language
  - Typically manual (without guarantee on correctness).
  - Precise approximation is obtained when the behaviour being removed is indeed not influencing the behaviour.

**Program slicing** is an example of an automated under-approximation that will lead to a precise abstraction.
Abstraction (3)

• Over-approximations (= abstract interpretation)
  - Maps sets of states in the concrete program to one state in the abstract program.
  - reduces the number of states, but increases the number of transitions, and hence the number of paths
• Examples
  - Type-base abstractions.
    - Replace int by Signs-abstraction (neg, pos, zero).
  - Predicate abstractions.
    - Replace predicates (conditions) in the program by boolean variables, and replace each instruction that modifies the predicate with a corresponding instruction that modifies the boolean.
• Automatic, conservative abstraction.

Abstraction (4)

• Data Type Abstraction
  - collapses data domains via abstract interpretation

Abstraction (5)

• Example of infeasible counterexample

Abstraction (6)

• Over-approximations (cont.)
  - Problem: elimination of spurious errors.
    - Abstract program has more behaviour. Therefore, when error is found in the abstract program, is that also an error in the original program?
    - In other words: over-approximations preserve correctness, but do not preserve errors.
    - Eliminating spurious errors is an active research area.
• JPF: heuristic search
  - First try to find a path to the error without non-deterministic choices (choose free path).
  - If there is not such a path, JPF will use simulation mode to find the divergence between the abstract and concrete path.
  - This divergence point is used to refine the abstraction.
Abstraction (7)

- Predicate Abstraction - idea:
  - map the concrete state space to an abstract state space, whose abstract states correspond to truth values of a set of boolean predicates
  - create abstract state-graph during model checking, or
  - create abstract transition system before model checking

Abstraction (8)

- Predicate Abstraction - example:

  **original program**
  ```java
  void foo() { 
    int i; 
    i=0; 
    while(even(i)) { 
      i++; 
    } 
  }
  ```

  **boolean program**
  ```java
  void foo() { 
    boolean p1, p2; 
    p1 = true; 
    p2 = true; 
    while(p2) { 
      p1 = p1?false:*; 
      p2 = !p2; 
    } 
  }
  ```

  * means either true or false, i.e. nondeterministic choice

Abstraction (9)

- Predicate Abstraction - how it works

  concrete statement Q: y:=y+1;
  what is the abstract statement w.r.t. the predicate B: {x=y}?

  1. weakest precondition: \(\text{wp}(Q,B)\) and \(\text{wp}(Q,!B)\)
     
     ```
     \text{wp}(y:=y+1, x_1=y_1) = x_0=y+1 \\
     \text{wp}(y:=y+1, x_1\neq y_1) = x_0\neq y+1 \\
     ```

     \(\text{wp}(\text{stat}, \text{post}) = \text{weakest precondition, such that after performing \text{stat}, \text{post certainly holds}}\)

  2. S(E): strengthening of \(\text{wp}(Q,B)\) and \(\text{wp}(Q,!B)\)

     ```
     S(x_1=y_1) = \text{false} \\
     S(x_1\neq y_1) = x_0=y_0 \\
     ```

     \(S(E)\) yields the best (conjunction of) predicates that implies \(E\), i.e. map the \(\text{wp}\) in terms of the known predicates.

  3. abstract statement now becomes:

     ```
     B := B ? \text{false} : *; \\
     ```

     * means either true or false

Static Analysis (1)

- Static analysis of programs consist of analysing programs without executing them. E.g.

  - data flow analysis
  - set and constraint resolution
  - abstract interpretation
  - theorem proving

  **JPF:** model checking of (Java) programs will not be tractable if partial order reductions are not supported by the model checker.

  **Goal w.r.t. model checking:** reduce the state space.

  **JPF uses following static analysis techniques:**

  - static slicing: smaller, but functionally equivalent program
  - partial evaluation: propagates constants + simplifies expressions
  - partial order computation: eliminates unnecessary interleavings does not change the size of the program
Static Analysis (2)

- Slicing
  - Idea is to eliminate statements that are not relevant to the property one wants to verify.
  - Static backward program slice
    = set of expressions and statements that may affect the value of a chosen variable reference during execution

- JPF implements property-directed slicing
  - Slicing criterion is generated automatically from the property that is being verified
  - Slicing algorithm automatically finds all parts of the program that might influence the variables mentioned in the property

JPF uses Bandera’s slicing tool.

Static Analysis (3)

Property to check: \( \Box (\text{isFull()} \Rightarrow (\text{head} = \text{tail})) \)

Static Analysis (4)

- Slicing (concurrent) programs entails the computation of a set of program dependencies.
  - Traditional dependencies (dataflow analysis)
    - Data: assignment
    - Control: if, while
    - Divergence: non-terminating loop
  - Concurrent dependencies
    - Interference: definition of shared variables can reach across threads
    - Synchronisation: variable is defined inside some critical section, the inner-most locking should be preserved
    - Ready dependence: statement \( n \) is dependent on statement \( m \) if \( m \)'s failure to complete can block the thread containing \( n \).
**Runtime analysis**

- Runtime analysis
  - executing a program once
  - observing the generated execution trace
  - use information to predict whether
    - different execution traces may violate some properties
    - whereas the generated execution trace itself does not have to violate these properties
- JPF supports two algorithms for runtime analysis
  - data race detection
  - deadlock detection
- runtime analysis ≠ runtime monitoring
  - certain user-defined properties are monitored on runtime

**Data Race Detection (1)**

- data race
  - two concurrent threads simultaneously access a shared variable
  - one of the accesses is a write
- program is data race free
  - if for every variable there is a nonempty set of locks that all threads own when they access the variable
- JPF implements Eraser algorithm [Savage et. al. 1997].

**Data Race Detection (2)**

- Eraser algorithm
  ```
  set(x) - locks active when threads access the variable x
  set(t) - locks taken by the thread t at any time
  Whenever a thread t accesses the variable x, the set(x) is refined as follows:
  if (x is accessed for the first time)
  then set(x) = set(t)
  else set(x) = set(x) \cap set(t)
  A race condition is possible if set(x) ever becomes empty.
  ```
- Disadvantage: too many warnings, e.g.
  - shared variables are often initialized without holding the lock.
  - if a thread creates an object which is read by several other threads (but no-one is writing after initialisation)

**Data Race Detection (3)**
Data Race Detection  (4)

- Implementation of the Eraser algorithm in JPF
  - JVM\textsuperscript{PF} accesses the bytecodes of the Java program
  - each bytecode has a Java class associated with
  - JVM\textsuperscript{PF} extends these classes
    - add \texttt{execute} method
      captures semantics of bytecode instruction
  - selected bytecodes are now instrumented to update the sets \texttt{set(x)} and \texttt{set($\ell$)}:
    - \texttt{GETFIELD}, \texttt{PUTFIELD}: read/write object fields
    - \texttt{MONITORENTER}, \texttt{MONITOREXIT}: updates of the lock sets of the accessing threads
    - \texttt{INVOKEVIRTUAL}, \texttt{INVOKESTATIC}: synchronized methods

Deadlock Detection  (1)

- classical deadlock situation
  - two threads share locks and attempt to take the locks in different order
  - not always an error; there might be a gate lock, which has to be acquired before any of the other locks are taken

```
Thread 1:
synchronized (L1) {
  synchronized (L3) {
    synchronized (L2) {
      synchronized (L4) {
        synchronized (L3) {
          synchronized (L2) {
            synchronized (L4) {
              synchronized (L3) {
                \texttt{potential deadlock}
              }
            }
          }
        }
      }
    }
  }
}
```

```
Thread 2:
synchronized (L4) {
  synchronized (L3) {
    synchronized (L2) {
      synchronized (L1) {
        \texttt{L1: gate lock}
      }
    }
  }
}
```

Runtime Analysis + Model Checking

- Runtime analysis:
  - already useful information (to programmer) as stand-alone tools
- Guide the model checker
  - run the program in simulation mode
  - obtain set of warnings about data races and lock order conflicts
  - threads causing the warnings are stored in a race window
  - extend the race window to include threads that create or otherwise influence the threads in the original window

  \textit{Purpose is now to get a small self-contained sub-system containing the race window, which can meaningfully be model checked.}
**Software Verification**

**Overview**

Gerard J. Holzmann.  
From Code to Models.  

Willem Visser, Klaus Havelund, Guillaume Brat,  
SeungJoon Park and Flavio Lerda.  
Model Checking Programs.  

**SLAM**

http://research.microsoft.com/slam/

- Programs are represented as boolean programs.  
  • values are of type `bool`  
  • allows control non-determinism

**Software Model Checking (3)**

- Abstraction based  
  - An abstract, overapproximated model is analyzed using a model checker.

  - SLAM  
    [Ball & Rajamani 2002]  
    http://javapathfinder.sourceforge.net/

  - BLAST  
    [Henzinger, Jhala & Majumdar 2005]

  This type of model checking is now called CEGR (counter example guided abstraction refinement).
• Consider a spin lock, which works roughly as follows:
  - Try to acquire the lock using some atomic operation.
  - If the lock could not be obtained, keep trying (spinning) until you can.

Requirements of a spin lock:
- The lock may only be released when it is locked.
- The lock may only be acquired when it is not locked.

C2BP: transforms a C program into a boolean program

Requirements in SLIC

SLIC = Specification Language for Interface Checking

Theo Ruys SV 6 Software Verification

SLAM - Example (3)

C2BP: transforms a C program into a boolean program

SLAM - Example (4)

newton discovers additional predicates to refine the boolean program, by analysing the feasibility of paths in the C program.

Theo Ruys SV 6 Software Verification
Conclusion

- **FeaVer**: semi-automatic
  - uses translation table to map C code fragments onto Promela
  - uses SPIN’s bit-state hashing to find errors

- **JPf**: automatic
  - starts with the full, concrete (bytecode) Java program
  - uses numerous reduction techniques (symmetries, abstractions, static analysis, run-time analysis, etc.) to curb the state space explosion

- **SLAM**: automatic
  - starts with a very abstract view of the original C program
  - if unfeasible error is found, abstraction is refined