

Automatic Verification

Introduction

(Based on [Clarke et al. 1999])

Yih-Kuen Tsay

Dept. of Information Management
National Taiwan University

The Context

- 🌐 Computer systems are increasingly used in applications where failure is unacceptable: electronic commerce, air traffic control, medical instruments, etc.
- 🌐 Problem: **validation/verification of design/program correctness**
 - ☀️ A major challenge in developing complex systems
 - ☀️ Approaches to design validation/verification:
 - 👤 **Simulation and testing**: effective when there are many bugs, non-exhaustive, hopeless to scale up
 - 👤 **Formal verification**: exhaustive (find subtle bugs), hard to scale up
- 🌐 Approaches to formal verification:
 - ☀️ **deductive**: time-consuming, by experts
 - ☀️ **algorithmic** (automatic): computational limitation

Example 1: Peterson's Algorithm for Two Processes

```
/* P0 */
```

```
...
```

```
Q[0] := true;
```

```
TURN := 0;
```

```
await  $\neg Q[1] \vee TURN \neq 0$ ;
```

```
critical section;
```

```
Q[0] := false;
```

```
...
```

```
/* P1 */
```

```
...
```

```
Q[1] := true;
```

```
TURN := 1;
```

```
await  $\neg Q[0] \vee TURN \neq 1$ ;
```

```
critical section;
```

```
Q[1] := false;
```

```
...
```

Note: $Q[0]$ and $Q[1]$ are *false* initially.

Example 1: Peterson's Algorithm for Two Processes

```
/* P0 */
```

```
...
```

```
Q[0] := true;
```

```
TURN := 0;
```

```
await  $\neg Q[1] \vee TURN \neq 0$ ;
```

```
critical section;
```

```
Q[0] := false;
```

```
...
```

```
/* P1 */
```

```
...
```

```
Q[1] := true;
```

```
TURN := 1;
```

```
await  $\neg Q[0] \vee TURN \neq 1$ ;
```

```
critical section;
```

```
Q[1] := false;
```

```
...
```

Note: $Q[0]$ and $Q[1]$ are *false* initially.

Question: How can its correctness be verified?

Example 2: Which Day of the Year

```
year = ORIGIN;
while (days > 365) {
  if (isLeapYear(year)) {
    if (days > 366) {
      days -= 366;
      year += 1;
    }
  }
  else {
    days -= 365;
    year += 1;
  }
}
```

Example 2: Which Day of the Year

```
year = ORIGIN;
while (days > 365) {
  if (isLeapYear(year)) {
    if (days > 366) {
      days -= 366;
      year += 1;
    }
  }
  else {
    days -= 365;
    year += 1;
  }
}
```

Question: What's wrong with the program?

Example 3: The Collatz Conjecture





```
...  
int x = N; /* N is some positive integer. */  
while (x > 1) {  
    if (x % 2 == 0) {  
        x = x / 2;  
    }  
    else  
        x = 3 * x + 1;  
}  
...
```

Example 3: The Collatz Conjecture

```
...
int x = N; /* N is some positive integer. */
while (x > 1) {
    if (x % 2 == 0) {
        x = x / 2;
    }
    else
        x = 3 * x + 1;
}
...
```

Question: Will the while loop terminate?

Course Subjects

-  Model checking algorithms/tools
-  Classic/general decision procedures/tools
-  Reduction and abstraction techniques for scalability
-  Theoretical foundations

Model Checking

- 🌐 Main activity: determining if the specification is true of a (finite-state concurrent) system, i.e., *checking* if the system is a *model* of the specification
- 🌐 The process:
 - ☀️ **Modeling**: convert a design into a formal model
Main systems considered: *finite-state transition systems* (modeling digital circuits, communication protocols, etc.)
 - ☀️ **Specification**: state the properties that the design must satisfy
Typical specification languages: *propositional modal/temporal logics*
 - ☀️ **Verification**: is automatic ideally, but often involves human assistance in practice

Model Checking (cont.)

- 🌐 Advantages (over deductive verification methods):
 - ☀ Fully automatic
 - ☀ Providing counterexamples
- 🌐 Main obstacle: the **state-explosion** problem (the number of states grows exponentially with the number of components or variables)
- 🌐 Became practically viable with **symbolic** encoding
- 🌐 Has been most successful in verifying hardware and communication protocols
- 🌐 Commercial model checking tools in the market

Early History of Model Checking

- 🌐 Introduction of temporal logic to *concurrent* programs [Pnueli 1977]
- 🌐 Temporal logic model checking algorithms [Clarke and Emerson 1981] [Queille and Sifakis 1982]
- 🌐 Linear-time algorithm for CTL [Clarke, Emerson, and Sistla 1983]
- 🌐 PSPACE-complete for LTL [Sistla and Clarke 1985][Pnueli and Lichtenstein 1985]
- 🌐 PSPACE-complete for CTL* [Clarke, Emerson, and Sistla 1983]
- 🌐 Automata-theoretical approach: model checking as language containment [Aggarwal, Kurshan, and Sabnani 1983][Vardi and Wolper 1986]

Alleviating State Explosion

- 🌐 Symbolic algorithms [McMillan 1993]: concise representations and efficient manipulation of boolean functions by *binary decision diagrams* [Bryant 1986]
- 🌐 Partial order reduction [Katz and Peled 1988][Valmari 1990][Godefroid 1990][Peled 1994]: equivalent computations from different orderings of independent events need not be distinguished; sufficient to keep just one representative computation
- 🌐 Other techniques
 - ☀ Abstraction
 - ☀ Compositional reasoning
 - ☀ Symmetry reduction
 - ☀ Induction (for infinite families of systems)