Generation of Concurrency Control Code using Discrete-Event Systems Theory

Christopher Dragert, Juergen Dingel, Karen Rudie

Presented by David Gerhard
Motivation

- Optimal usage of today's Multi-core systems requires parallel executable Software
- Efficient and correct concurrent source code is hard to write and debug

→ facilitate development of concurrent source code
Introduction

• Automatic generation of concurrency control code
  • Input
    – Source code without concurrency control
    – Informal specification of the desired concurrent behaviour
  • Output
    – Source code with concurrency controls
Introduction

• Aims
  • No Deadlocks
  • No Starvation
  • Minimally restrictive

→ using control theory (Discrete-event System)
DES - Supervisor-Plant

- System(Plant) modelled as a finite-state automaton (FSA)
  - Transitions in FSA are called events
    - Events can be controllable or uncontrollable
- Supervisor modelled as a FSA
  - Enables or disables controllable events
Process Overview

- Uncontrolled Source Code
- Informal Specification
- Controlled Source Code
Process Overview

Uncontrolled Source Code

Event-Marked Source Code

Informal Specification

Formal Specification

DES Supervisor

DES Plant

Controlled Source Code
Relevant Events

• Find relevant events for concurrency control
  • Example
    - Enter/Exit Critical Section → relevant
    - Accessing shared variable → non-relevant

• Mark events in the source code

• Differentiate controllable and uncontrollable events
Example

- 5 Threads with dependencies
  - 1 → 2,3,4
  - 5 → 4

Code example for Thread 3:
```java
public void run() {
    // relevant event: T3-start
    System.out.println(id);
    doWork();
}
```
DES Plant

- Build a Finite-state automaton (FSA) representing all possible event sequences for each Thread
  - Build control-flow graph (CFG) from source code
  - Reduce CFG
    - All relevant events remain
    - All non-relevant events important for CFG structure remain
Example

Code example for Thread 3:

```java
public void run() {
    // relevant event: T3-start
    System.out.println(id);
    doWork();
}
```

![Diagram showing the flow of the code example]
Process Overview

- Uncontrolled Source Code
- Informal Specification
- Event-Marked Source Code
- Formal Specification
- DES Supervisor
- DES Plant
- Controlled Source Code
Formal Specification

- Specifies the allowed subset of event sequences
  - FSA for each restriction
  - Only restrictive
Example

1. T1-finish, T2-start, T3-start, T4-start, T5-finish
2. T1-finish, T2-start, T3-start, T4-start, T5-finish
DES Supervisor

- Only one supervisor (simplification)
- Different FSA's need to be combined into a monolithic specification
  - Scalability issue
Process Overview

Uncontrolled Source Code → Event-Marked Source Code

Informal Specification → Formal Specification

DES Supervisor

DES Plant

Controlled Source Code
Code Generation

• Supervisor needs to block non allowed controllable events
  • Generates a Semaphore for each controllable event set to the initial state
  • Supervisor state changing function enables/disables controllable events
Example – Thread 3

// relevant event: T3start

while (true) {
    if(Synchronizer.stateChangeTest("T3start", Synchronizer.T3start)) {
        break;
        Synchronizer.T3start.acquireUninterruptibly();
        Synchronizer.T3start.release();
    }
}
Example – Supervisor I

public static synchronized Boolean stateChangeTest(String event, Semaphore eventBlocker) {
    if (!(eventBlocker == null)) {
        if (!eventBlocker.tryAcquire()) {
            return false;
        }
        eventBlocker.release();
    }
    changeSupervisorState(event);
    return true;
}
private static void changeSupervisorState(String event) {
    if (event.equals("T1finish")) {
        switch(Synchronizer.stateTracker) {
            case(0):
                Synchronizer.T3start.release();
                Synchronizer.T2start.release();
                Synchronizer.stateTracker = 1;
                break;
            case(1):
        ...
    ...
Verification

• Discrete-event theory is proven correct and non-blocking
• Formal proof for algorithm is still needed
• Modelchecker (Java Pathfinder)

• Input not reliable!
Limitations

- No dynamic threads generation allowed
- Monolithic supervisor not very efficient
- Starvation not properly addressed
Conclusion

- DES theory can be applied to concurrency
- Chosen FSA-based version of DES not expressive enough
- Further work needed
Questions