MaC
Monitoring and Checking at Runtime

Presented By
Usa Sammapun
CIS 700 Oct 10, 2005
What is MaC?

- A verification technique
  - Goal: **Ensure a software program runs correctly**

- To understand software verification
  - Know how software is developed
  - Know how software is verified
Software Development Process

Requirement and Properties

- What program should do
  - When AIBO dog walks, it must not fall

- Informal (English) \(\rightarrow\) Formal (Logic, FSM)
Software Development Process

Design Specification and Analysis

- How program fulfill requirements
  - AIBO dog coordinates his 4 legs
- Formal modeling (UML, FSM, Control theory)
- Analysis
  - Simulation
  - Verification (Model checking)
Software Development Process

◆ Implementation
  • Actual program  (AIBO dog walking program in C++)
  • Verification & Validation
    – Testing
    – Runtime verification

```
main() {
  int leg1, leg2;
  int leg3, leg4;
  ...
}
```
Verification

- Design
  - Model Checking

- Implementation
  - Testing
  - Runtime Verification

MaC
Monitoring and Checking At Runtime
Verification

► Design
  – Model Checking

► Implementation
  – Testing
  – Runtime Verification
Model Checking

- **Given**
  - Requirement & Properties
  - Model

- **Verify**
  - Explore all paths
  - Violate requirement ??

Walk → !(Fall)

- `all-leg-down`
- `walk`!

- `leg1-up, leg4-up`
- `walk & !fall`

- `leg2-up, leg3-up`
- `walk & !fall`

- `leg1-up, leg2-up`
- `walk & fall`
Model Checking

- **Given**
  - Requirement & Properties
  - Model

- **Verify**
  - Explore all paths
  - Violate requirement ??

Given:
- Requirement & Properties
- Model

Verify:
- Explore all paths
- Violate requirement ??
Model Checking - GOOD

- Rigorous and Formal
  - Based on Mathematics

- Complete
  - Explore all paths
Model Checking - PROBLEM

- **Check Design, not implementation**
  - What if implementation does not follow model?

- **Not scalable**
  - What if the program is HUGH?
    - Explore all paths might not be feasible
Verification

► Design
  – Model Checking

► Implementation
  – Testing
  – Runtime Verification
Testing

► We’ve seen it
  – Run actual program with different inputs
  – See if outputs are what we want

► Ex. AIBO
  – Run AIBO dog
  – See whether or not AIBO dog falls

► Good
  – Check directly the implementation
Testing – PROBLEM

► Not rigorous, Not formal
  – Possibly random inputs

► Not complete
  – What if bugs never show up during test ??
  – What if it’s not AIBO, but a heart device !?!
Verification

- Design
  - Model Checking

- Implementation
  - Testing
  - Runtime Verification

MaC
Monitoring and Checking At Runtime
Runtime Verification

- **Given**
  - Requirement & Properties
  - Implementation

- Ensures the current program execution follows its formal requirements
Runtime Verification

1. Specify formal requirements (Walk $\rightarrow$ !Fall)
Runtime Verification

- Rigorous and Formal
- Done at implementation
- Not complete
  - Guarantee for current execution
Runtime Verification

- **MaC**
  - Monitoring and Checking at Runtime

- **Components**
  - MaC verifier
  - MaC formal language
MaC Verifier and Language

1. Which program info to probe
2. How to map info to properties

System Properties

Where / when to steer

Program

MaC Specification

PEDL

MEDL

SADL

MaC Compiler

Instrumented Program

MaC Verifier

Event Recognizer

Checker

Injector
Abstract Information

► To capture roughly and abstractly what the program is doing

► Events
  – Instantaneous incidents
  – such as variable updates update(position)

► Conditions
  – Proposition about the program that may be true/false/undefined for a duration of time
  – such as position < 50

position = 60  position = 55  position = 40  position = 55

\[\text{position < 50} \quad \text{false} \quad \text{true}\]

\[s_1 \quad s_2 \quad s_3 \quad s_4\]
Events

- **e** - variable update, start/end method
- **e1 || e2** - or
- **e1 && e2** - and
- **start(c)** - instant when condition c becomes true
- **end(c)** - instant when condition c becomes false
- **e when c** - e occurs when condition c is true
Conditions

- Conditions interpreted over 3 values: true, false and undefined.

- \( c \) - boolean expression
- \( !c \) - not c
- \( c_1 \lor c_2 \) - or
- \( c_1 \land c_2 \) - and
- \( c_1 \rightarrow c_2 \) - imply
- \( \text{defined}(c) \) - true when c is defined
- \([e_1, e_2)\) - interval
MaC Language

► PEDL
  – How execution information transform into events and conditions

► MEDL
  – Specify properties using events and conditions
**Railroad Crossing Property**: - If train is crossing, then gate must be down
- Train is crossing when position is between 30 and 50

**Abstraction**
- When train position is between 30 and 50
  - When gate starts/ends being down

**Check**
- If train is crossing, then gate must be down

**Violation**
- When gate is down

---

Java Program

- position = 0
- position = 20
- position = 40
- position = 55
- position = 60

- Gate.down()
- Gate.up()

PEDL

- startGD
- endGD
- cross

MEDL

- cross
- gateDown
- Violation
class Train {
    int position;
    main() {
        position = 0;
        position = 20;
        position = 40;
        position = 55;
    }
}

class Train {
    int position;
    main() {
        position = 0;
        send(x,0);
        position = 20;
        send(x,20);
        position = 40;
        send(x,40);
        position = 55;
        send(x,55);
    }
}

Sent to Event Recognizer:
[ (position,0), (position,20),
(position,40), (position,55) ]
MaC Language - PEDL

**Railroad Crossing Property**: - If train is crossing, then gate must be down
- Train is crossing when position is between 30 and 50

**Abstraction**
- When train position is between 30 and 50
- When gate starts/ends being down

**Java Program**
```java
export event startGD, endGD;
export condition cross;

monobj Train.position;
monmeth Gate.up();
monmeth Gate.down();

condition cross = (30 < RRC.position) && (RRC.position < 50);
event startGD = endM(Gate.down());
event endGD = startM(Gate.up());
```

**PEDL**

- position = 0
- position = 20
- position = 40
- Gate.down()
- position = 55
- Gate.up()
- position = 60
MEDL – Property Language

- Composed using
  - Events
  - Conditions
  - Connectives

- Properties
  - **Alarms:** events that must never occur
    \[\text{alarm} \text{ elevator} = \text{door\_open} \quad \text{when} \quad !\text{floor\_level}\]

  - **Safety Properties:** conditions that must always hold true
    \[\text{property} \text{ rail\_road} = \text{train\_cross} \quad \rightarrow \quad \text{gate\_down}\]
**Railroad Crossing Property**: - If train is crossing, then gate must be down
  - Train is crossing when position is between 30 and 50

**Abstraction**
- When gate is down

**Check**
- If train is crossing, then gate must be down

```plaintext
import event startGD, endGD;
import conditions cross;

condition gateDown = [startGD, endGD);
property safeRRC = cross -> gateDown;
```
Current Work

► Timing properties

► Probabilistic properties

► Dynamic MaC

► Steering using control theory
Quantitative Properties

- Time bound interval: $[e_1, e_2] \leq d$, $[e_1, e_2) < d$, $[e_1, e_2] = d$

Diagram:

- $[e_1, e_2] \leq d$
- $[e_1, e_2] \leq d$
- $[e_1, e_2] \leq d$
- $[e_1, e_2] \leq d$

$\leq d$

- $[e_1, e_2] \leq d$
- $[e_1, e_2] \leq d$

$> d$
Example

- A real-time task $T$ must finish within 100 time units
  - $startT$ – event when task $T$ starts executing
  - $endT$ – event when task $T$ finishes executing

$[startT, endT) \leq 100$