Runtime Assurance Based on Formal Specifications

and

Purify
Administrivia

• Homework 2
  - due next Thursday (Feb 12)
• Blog – ok?
• Presentations
  - first presentation is today
  - let me know if there’s a particular area or paper you’d like to present
  - I’m putting a list of additional possibilities on the course web page
Administrivia - calendar

• **CMPS 280G** Thurs 2-3:45pm, Crown 105
  - Feb 12: Arnaud Venet and Guillaume Brat
    • NASA’s C Global Surveyor
  - Feb 19: Willem Visser
    • Java PathFinder model checker
  - Feb 26: Klaus Havelund from NASA Ames

• **Interview candidates**
  - *Monday March 1*\(^{st}\): Benjamin Pierce
    • distributed debugging
  - *Thursday March 4*\(^{th}\): Ranjit Jhala
    • BLAST model checker for Linux device drivers
Big Picture of *Runtime Assurance* ...

- Real time systems have intended behaviors
  - really, specifications

- Check dynamically for violations of safety specification
  - e.g. fail-stop, or to alert human operator

- Could write code for this
  - in a regular language
  - in a domain-specific language (this paper)
Design

• Two pieces

• Recognize primitive events
  - Interesting things that happen in the program
  - Specified by a PEDL script
    • Primitive Event Definition Language

• Check requirements
  - Specified by a MEDL specification
    • Meta-Event Definition Language
Why Two Languages?

• Convenient to write low-level predicates in the target programming language

• But higher-level specifications will need to be in some kind of logic

• Separating the two makes the system more portable
Monitoring and Checking (MaC) Framework

Java Program

Automatic Instrumentation

System Filter

Event Recognizer

Run-time Checker

Requirement Spec

Monitoring Script

low level PEDL

high level description

Automatic Translation

Automatic Translation

low-level events

high-level events

Input

Run-time Process

Static Process

Human
Primitives

• Examples
  
  TooFast = Train.calculatePosition().trainspeed > 100
  
  InMethod(f)
  
  MethodEnter(f)
  
  MethodExit(f)
Conditions and Events

• Examples

   TooFast = Train.calculatePosition().trainspeed > 100
   InMethod(f)
   MethodEnter(f)
   MethodExit(f)

• An event happens at a moment in time
Conditions and Events

• Examples
  
  TooFast = Train.calculatePosition().trainspeed > 100
  
  InMethod(f)
  
  MethodEnter(f)
  
  MethodExit(f)

• A condition is true for intervals of time
Conditions and Events

• Examples
  
  \[ \text{TooFast} = \text{Train.c}\text{alculatePosition()}.\text{trainspeed} > 100 \]
  \[ \text{InMethod(f)} \]
  \[ \text{MethodEnter(f)} \]
  \[ \text{MethodExit(f)} \]

• Two events characterize every condition
  - When the condition first becomes true
  - When the event first becomes false
Definitions

• Examples
  TooFast = Train.calculatePosition().trainspeed > 100
  InMethod(f)
  MethodEnter(f)
  MethodExit(f)

• Primitives can be named
  - For use in MEDL specifications
Time

• Examples

  TooFast = Train.calculatePosition().trainspeed > 100
  InMethod(f)
  MethodEnter(f)
  MethodExit(f)

• Every primitive event comes with a timestamp
  - To enable checking of real-time properties
Undefined Conditions

• Examples
  TooFast = Train.calculatePosition().trainspeed > 100
  InMethod(f)
  MethodEnter(f)
  MethodExit(f)

• Conditions may not be true or false
  - Because of uninitialized data
  - Add explicit “undefined” value
  - Add primitive condition undefined(c)
MEDL

• Primitive events & conditions can be combined
  - Note no negation for events

\[ e ::= e_p \]
\[ \text{start}(c) \]
\[ \text{end}(c) \]
\[ e \&\& e \]
\[ e \mid\mid e \]
\[ e \text{ when } c \]

\[ c ::= c_p \]
\[ \text{defined}(c) \]
\[ [e,e) \]
\[ !c \]
\[ c \&\& c \]
\[ c \mid\mid c \]
\[ c \uparrow c \]
Semantics

- Primitive events/conditions are tested at specified points
  - Part of PEDL script
  - Designed to be cheap
    - Primitives depend only on current state

- MEDL semantics is 3-valued logic
  - true, false, undefined (for conditions)
Auxiliary Variables

• Sometimes need stateful monitoring
  - E.g., a count of certain events

• MEDL allows definition of auxiliary variables

\[ e_1 \text{, } \text{count}_{e_1} := \text{count}_{e_1} + 1 \]

• A bit ugly
  - Auxiliary variables/computations in MEDL expressed in host programming language
Example: A Train Crossing Controller

/* export to MEDL */
Export event OpentGate, CloseGate
Expert condition GateDown

/* monitoring only at these points */
Monitored Entities:
  void GateController.open();
  void GateController.close();
  int GateController.gatePosition;
Example: A Train Crossing Controller (Cont.)

/* define a condition */
CondDef:
  Cond GateDown = (GateController.gatePosition == GateController.GATE_DOWN)

/* define events */
EventDef:
  Event OpenGate = MethodEnter(GateController.open());
  Event CloseGate = MethodEnter(GateController.close());
Example: A Train Crossing Controller (Cont.)

/* MEDL spec */
Import event OpenGate, CloseGate;
Import condition GateDown;
AuxVarDecl: float startClose, currentTime;

Cond GateClosing =
[ CloseGate when !GateDown, OpenGate || start(GateDown))
†
    startClose + 30 > currentTime
Opinions

• Could write monitoring code directly in Java
  - but would complicate (+ break?) monitored code
  - and Java monitors might be awkward to write/read

• A runtime monitoring toolkit is a good idea
  - separates code from specification
  - clearer, readable specification
  - leads to programmers writing specifications?

• This paper represents a start
  - Need more expressiveness and applications
Purify
Purify

- Purify detects
  - reads of uninitialized memory
  - accesses to deallocated memory
  - accesses out of bounds
- Memory instrumentation via memory map
  - 2 bits per byte of memory
    - unallocated, uninitialized, initialized
    - “red zone”
  - Purify substitutes its own malloc
- Each load/store instrumented to test/set bits
Welcome to Purify

Please select your first step...

- Run
- Open
- Continue

Did you know?... Using Purify consistently at every stage in your development process helps you deliver the highest quality software applications.

© Copyright 1992-1998 Rational Software Corporation

Next tip