Testing and Diagnosis

16.415/6.141 – Robotics: Science and Software
February 27th 2008

Today's Lecture

- Introduction to testing
- Some basic principles of testing: preconditions, postconditions and invariants
- Automated testing in Java
- How to take the description of a system and model its operation using propositional logic

What You Should Know

- How and why to carry out black-box testing
- Some guidelines on how to write tests
- How to use JUnit to write automated tests
- How to detect a failure given a model and observations
- How to generate a set of diagnoses given a model and observations

The Verification & Validation Process

- Is a whole life-cycle process - V&V must be applied at each stage in the software process
- Has two principal objectives
  - The discovery of defects in a system
  - The assessment of whether or not the system is usable in an operational situation
- *Software inspections* Concerned with analysis of the static system representation in order to discover problems (static verification)
  - May be supplemented by tool-based document and code analysis
  - We won't do this today, but will revisit this topic later in the semester
- *Software testing* Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed
- Testing and debugging are distinct processes
  - Verification and validation is concerned with establishing the existence of defects in a program
  - Debugging is concerned with locating and repairing these errors
- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error

Cost of Testing

Normal SDLC			Testing SDLC	
Accumulated Test Cost	Accumulated Errors/1000 LOC		Accumulated Errors/1000 LOC	Accumulated Test Cost
		Requirements 20 Errors Cost of Detection: 1		
		Design 20 Errors Cost of Detection: 1		
		Code 20 Errors Cost of Detection 1		
		Test 80% Error Reduction Cost of Error: 10		
		Production 100% Error Reduction Cost of Error: 100		

Verification and Validation of Modern Software Intensive Systems - Schulmeyer, G. Gordon

Cost of Testing

Normal SDLC			Testing SDLC	
Accumulated Test Cost	Accumulated Errors/1000 LOC		Accumulated Errors/1000 LOC	Accumulated Test Cost
0	20	**Requirements** 20 Errors Cost of Detection: 1	(Testing: 50% Error Reduction) 10	10

Verification and Validation of Modern Software Intensive Systems - Schulmeyer, G. Gordon

Cost of Testing

Normal SDLC			Testing SDLC	
Accumulated Test Cost	Accumulated Errors/1000 LOC		Accumulated Errors/1000 LOC	Accumulated Test Cost
0	20	**Requirements** 20 Errors Cost of Detection: 1	(Testing: 50% Error Reduction) 10	10
0	40	**Design** 20 Errors Cost of Detection: 1	(Testing: 50% Error Reduction) 15	25

Verification and Validation of Modern Software Intensive Systems - Schulmeyer, G. Gordon

Cost of Testing

Normal SDLC			Testing SDLC	
Accumulated Test Cost	Accumulated Errors/1000 LOC		Accumulated Errors/1000 LOC	Accumulated Test Cost
0	20	**Requirements** 20 Errors Cost of Detection: 1	(Testing: 50% Error Reduction) 10	10
0	40	**Design** 20 Errors Cost of Detection: 1	(Testing: 50% Error Reduction) 15	25
0	60	**Code** 20 Errors Cost of Detection 1	(Testing: 50% Error Reduction) 18	42

Verification and Validation of Modern Software Intensive Systems - Schulmeyer, G. Gordon

Cost of Testing

Normal SDLC			Testing SDLC	
Accumulated Test Cost	Accumulated Errors/1000 LOC		Accumulated Errors/1000 LOC	Accumulated Test Cost
0	20	**Requirements** 20 Errors Cost of Detection: 1	(Testing: 50% Error Reduction) 10	10
0	40	**Design** 20 Errors Cost of Detection: 1	(Testing: 50% Error Reduction) 15	25
0	60	**Code** 20 Errors Cost of Detection 1	(Testing: 50% Error Reduction) 18	42
480	12	**Test** 80% Error Reduction Cost of Error: 10	4	182

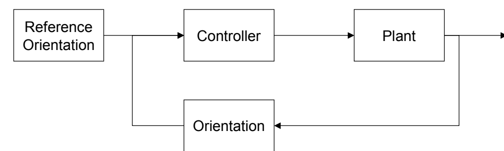
Verification and Validation of Modern Software Intensive Systems - Schulmeyer, G. Gordon

Cost of Testing

Normal SDLC			Testing SDLC	
Accumulated Test Cost	Accumulated Errors/1000 LOC		Accumulated Errors/1000 LOC	Accumulated Test Cost
0	20	**Requirements** 20 Errors Cost of Detection: 1	(Testing: 50% Error Reduction) 10	10
0	40	**Design** 20 Errors Cost of Detection: 1	(Testing: 50% Error Reduction) 15	25
0	60	**Code** 20 Errors Cost of Detection 1	(Testing: 50% Error Reduction) 18	42
480	12	**Test** 80% Error Reduction Cost of Error: 10	4	182
1680	0	**Production** 100% Error Reduction Cost of Error: 100	0	582

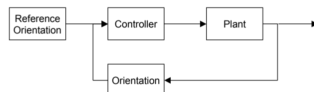
Verification and Validation of Modern Software Intensive Systems - Schulmeyer, G. Gordon

A Simple Embedded System



- How do we verify that our controller is working correctly?

Program Testing



- Types of Testing
  - Defect testing: Tests designed to discover system defects.
    - A successful defect test is one which reveals the presence of defects in a system.
  - Statistical testing: Tests designed to reflect the frequency of user inputs. Used for reliability estimation.
- A successful test is a test which discovers one or more errors
- Can reveal the presence of errors not their absence
- The most commonly used validation technique for non-functional requirements
- Should be used in conjunction with static verification

Implementing Tests

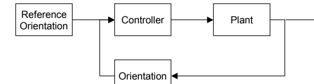
```
public class Pose {
  public double x, y, theta;

  public void updateHeading(double deltaTheta) {
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    return theta;
   }
}
```

Implementing Tests

```
public class Pose {
  public double x, y, theta;

  public void updateHeading(double deltaTheta) {
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    return theta;
  }

  public static void testUpdateHeading() {
    Pose p = new Pose(Math.random()*100, Math.random()*100, Math.random()*2*Math.PI);
    double deltaTheta = Math.random()*2*Math.PI);
    p.updateHeading(deltaTheta);
    assert(p.theta <= Math.PI);
    assert(p.theta > -Math.PI);
  }
}
```

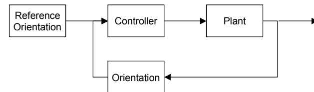
An external test to ensure that theta meets our bounds.

Test Data and Test Cases



- *Test data* Inputs which have been devised to test the system
- *Test cases* Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification
- Testing should be:
  - Repeatable:
    - If you find an error, you'll want to repeat the test to show others
    - If you correct an error, you'll want to repeat the test to check you did fix it
  - Systematic
    - Random testing is not enough
    - Select test sets that cover the range of behaviors of the program
    - are representative of real use
  - Documented
    - Keep track of what tests were performed, and what the results were

Preconditions, Postconditions and Invariants



- Preconditions/postconditions and invariants are commonly used in "design-by-contract" engineering
- Precondition - what must be true when a method is invoked. When a precondition fails, the method invoker has a fault.
- Postcondition - what must be true after a method completes successfully. When a postcondition fails, the method has a fault **or** the precondition was not met.
- Class Invariant - what must be true about each instance of a class after construction and after every method call. Also must true for static methods when there is no object of the class created. When an invariant fails, a fault could exist with the method invoker or the class itself.
- Another common kind of invariant is internal – conditions in the implementation we know must always hold

Implementing Preconditions

```
public class Pose {
  private double x, y, theta;
/**
  * Updates the heading.
  *
  * @param  deltaTheta heading change in radians.
  * @throws IllegalArgumentException if theta < -PI or
  * rate >= PI.
  */
  public void updateHeading(double deltaTheta) {
    if (deltaTheta < -Math.PI || deltaTheta >= Math.PI)
      throw new IllegalArgumentException("Invalid heading change: " + deltaTheta);
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    assert result >= -Math.PI && result < Math.PI : this;

    return theta;
}`
```

We have explicit enforcement of the precondition here, but we would also write an external test to ensure this precondition is being enforced.

Implementing Preconditions

```
public class Pose {
  private double x, y, theta;
/**
  * Updates the heading.
  *
  * @param  deltaTheta heading change in radians.
  * @throws IllegalArgumentException if theta < -PI or
  * rate >= PI.
  */
  public void updateHeading(double deltaTheta) {
    if (deltaTheta < -Math.PI || deltaTheta >= Math.PI)
      throw new IllegalArgumentException("Invalid heading change: " + deltaTheta);
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    assert result >= -Math.PI && result < Math.PI : this;

    return theta;
  }
  public static void testUpdateHeading() {
    Pose p = new Pose(Math.random()*100, Math.random()*100, Math.random()*2*Math.PI);
    double deltaTheta = 4*Math.PI;
    try {
      p.updateHeading(deltaTheta);
      assert(false);
    }
    catch (Exception e) { }
  }
}`
```

The test only succeeds if an exception is thrown before this point.

Guards

- Preconditions, postconditions and many internal invariants are properties that you can test in the method body itself. These internal tests we call "guards".
- We can also write external "black-box" tests to make sure the guards are upheld
- Including postcondition and internal invariant tests in the method body is part of a larger practice known as "defensive programming"
- Writing explicit tests for postconditions and invariants is somehow more "intuitive": you are checking to make sure the method worked correctly and the postconditions and invariants hold for every method
- There is an issue here with preconditions: you want to make sure that not only does the method accept reasonable arguments, but you want to test for failure of violated preconditions.
- In many cases, testing involves ensuring an exception is thrown.

Implementing Postconditions

```
public class Pose {
  private double x, y, theta;
/**
  * Updates the heading.
  *
  * @param  deltaTheta heading change in radians.
  * @throws IllegalArgumentException if theta < -PI or
  * rate >= PI.
  */
  public void updateHeading(double deltaTheta) {
  // Test precondition
    if (deltaTheta < -Math.PI || deltaTheta >= Math.PI)
      throw new IllegalArgumentException("Invalid heading change: " + deltaTheta);
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    assert result >= -Math.PI && result < Math.PI : this;

    return theta;
}
```

We have explicit enforcement of the postcondition here, but we would also write an external test to ensure this postcondition is being enforced.

Class Invariants

```
public class Pose {
  private double x, y, theta;
/**
  * Updates the heading.
  *
  * @param  deltaTheta heading change in radians.
  * @throws IllegalArgumentException if theta < -PI or
  * rate >= PI.
  */
  public void updateHeading(double deltaTheta) {
  // Test precondition
    if (deltaTheta < -Math.PI || deltaTheta >= Math.PI)
      throw new IllegalArgumentException("Invalid heading change: " + deltaTheta);
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    assert theta >= -Math.PI && theta < Math.PI : this;

    return theta;
}
```

This post-condition could be modelled as class invariant in other methods and the constructor. Can we write external tests to ensure that it holds after all method calls?

Internal Invariants

```
if (i % 2 == 0) {
  ...
} else { // i % 2 == 1?
   ...
}

switch(parity) {
  case Parity.EVEN:
      ...
      break;
  case Parity.ODD:
      ...
      break;
}

void method() {
  for (...) {
    if (...)
      return;
    }
  // We should never be here
}
```

This switch statement contains the (incorrect) assumption that parity can have one of only two values. To test this assumption, you should add the following default case:

```
   default:
         assert false : parity;
```

Equivalence Partitioning

- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition
- Example:
  - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are <10,000, 10,000-99,999 and >99,999
  - Choose test cases at the boundary of these sets: 9999, 10000, 99999, 100000
  - Consider adding additional cases: 50000? -1? 0? Others?
- Input partitions:
  - Inputs which conform to the preconditions
  - Inputs where a pre-condition does not hold
  - Edge cases
- Other guidelines for preconditions
  - Test software with arrays which have only a single value
  - Use arrays of different sizes in different tests
  - Derive tests so that the first, middle and last elements of the array are accessed
  - Test with arrays of zero length

Problem Statement

- A program accepts as input three integers which it interprets as the lengths of sides of a triangle. It reports whether the triangle is equilateral, isosceles, or scalene (neither equilateral nor isosceles).

  ```
  public static int type(int a, int b, int c);
  ```

- How many tests are there?

Test Cases

Test Cases

- Correctness of Input
  - a = b = c = 0
  - One of the sides is zero
  - a or b or c is a non-integer
  - Less than three inputs
  - More than three inputs
- Sum of any two sides is greater than the third
  - (a+b = c), (a+c = b), (b+c = a)
  - (a+b < c), (b+c <a), (a+c <b)
  - One of (a+b > c), (a+c >b), (b+c >a)
- Type of Triangle
  - a ≠ b≠ c
  - a = b = c
  - (a ≠ b, b = c), (a ≠ b, a = c), (a = b, a ≠ c)

JUnit – Automated Test Harness

- A JUnit test consists of a `TestSuite` that has tests added to it.
- A `TestRunner` is then used to invoke every test in the suite.
  - Tests can be added by calling `TestSuite.addTest` with a `TestCase` class

```
TestSuite suite = new TestSuite();
suite.addTest(new TriangleTest("testScalene"));
```

This is the name of the method that we wish to use as a test.

  - or by passing a `TestCase` class into the `TestSuite` constructor

```
TestSuite suite = new TestSuite(TriangleTest.class);
```

- If you create the test suite using the constructor, then JUnit will use reflection to discover all the methods that begin with the letters “test”, and add them to the `TestSuite`.

JUnit – an example

```
import junit.framework.*;

public class TestTriangle extends TestCase {
  public TestTriangle(String name) {
    super(name);
  }
  public void testScalene() {
    int a = Math.random()+2;
    int b = a+3;
    int c = a+4;
    assertTrue(Triangle.type(a, b, c) == Triangle.Scalene);
  }

  public void testAllZero() {
    try {
      Triangle.type(0, 0, 0);
      fail();
    }
    catch (IllegalArgumentException e) {
    }
  }
  public static void main(String args[]) {
    TestSuite suite = new TestSuite(TestTriangle.class);
    junit.textui.TestRunner.run(suite);
  }
}
```

What's the problem here?
Should add a private Random field to TestTriangle and Initialize it in the constructor:
random = new Random(System.currentTimeMillis());

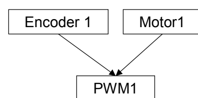
The random call changes to
Int a = random.nextInt(10)+2;

Implementation Issues

- Every `TestCase` derivative must have a constructor that takes a String argument, to specify which method to run as the test
- Repeated tests on similar objects can have repeated code
  - Use the `setUp/tearDown` fixtures
- Are there tests that we can’t easily automate?
  - GUIs, concurrent programs, distributed programs, hardware-dependent functionality, …

How Do We Reason About Complex Systems at a Commonsense Level?

Encoder 1 is going forward if and only if the PWM setting on Motor 1 is positive.



Is the diagnosis --- “the encoder is broken” closed --- consistent with the observations?

(mode(E1) = ok or mode(E1) = unknown) and
not (mode(E1) = ok and mode(E1) = unknown) and
(mode(E1) = ok implies
  (encoder(E1) = fwd if and only if motor(1) = on and PWM(1) = fwd))

Propositional Clauses: A Simpler Form

- Literal: A proposition or its negation
  - B, Not A
- Clause: A disjunction of literals
  - not A or B or E
- Conjunctive Normal Form: A conjunction of clauses
  - Φ = (A or B or C) and
(not A or B or E) and
(not B or C or D)
  - Viewed as a set of clauses

Reducing Propositional Formula to Clauses (CNF)

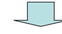
1) Eliminate IFF and Implies
- E1 iff E2 => (E1 implies E2) and (E2 implies E1)
- E1 implies E2 => not E1 or E2

2) Move negations in towards propositions using De Morgan's Theorem:
- Not (E1 and E2) => (not E1) or (not E2)
- Not (E1 or E2) => (not E1) and (not E2)
- Not (not E1) => E1

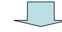
3) Move conjunctions out using Distributivity
- E1 or (E2 and E3) =>(E1 or E2) and (E1 or E3)

Reduction to CNF: Motor Example

Encoder 1 is going forward if and only if the PWM setting on Motor 1 is positive.



forward(E1) iff posPwm(M1)



not (forward(E1)) or posPwm(M1)
not (posPwm(M1)) or forward(E1)

Propositional Satisfiability

Input: A Propositional Satisfiability Problem is a pair <P, Φ >, where:
- P is a finite set of propositions.
- Φ is a propositional sentence on P
  - May be reduced to a set of clauses.

Output: True iff there exists an assignment of P consistent with Φ.

- This is an instance of a CSP:
  - Variables: Propositions
  - Domain: {True, False}
  - Constraints: Clauses that must be true
- The solution to the CSP is an interpretation (truth assignment to all propositions) such that all clauses are satisfied:
  - A clause is satisfied if and only if at least one literal is true.
  - A clause is violated if and only if all literals are false.

Clausal Backtrack Search

BT(Φ, A)

Input: A *cnf* theory Φ,
A partial assignment A to propositions in Φ

Output: A decision of whether Φ is satisfiable.

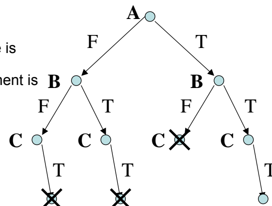
1. If a clause is violated, Return false;
2. Else If all propositions are assigned, Return true;
3. Else Q = some unassigned proposition in Φ
4. Return (BT(Φ, A[Q = True]) or
5. BT(Φ, A[Q = False])

Clausal Backtrack Search:

Backtrack Search
- Assign true or false to an unassigned proposition.
- Backtrack as soon as a clause is violated.
- Theory is satisfiable if assignment is complete.

Example:
- C1: Not A or B
- C2: Not C or A
- C3: Not B or C
- C4: C

where A = PWM is positive
B = encoder ticking forward
C = robot is moving

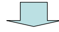


Model-based Diagnosis

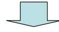
- What happens when no consistent model exists?
- Something must be broken.
- Let's perform diagnosis using additional variables:
  - X mode variables, one for each component c
  - D_c modes of component c = domain of $x_c \in X$
  - Y model variables and their domains
  - M(X, Y) model constraints
  - O observable variables $O \subseteq Y$
    - Partitioned into Inputs I and Responses R

Reduction to CNF: Motor Example

Motor 1 is working correctly implies that (encoder 1 is going forward if and only if the PWM setting on Motor 1 is positive).



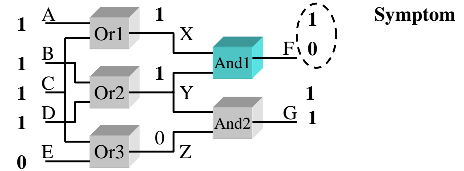
(ok(M1) implies (forward(E1) iff posPwm(M1))



not (ok(M1)) or not forward(E1) or pwm(M1)
not (ok(M1)) or not posPwm(M1) or forward(E1)

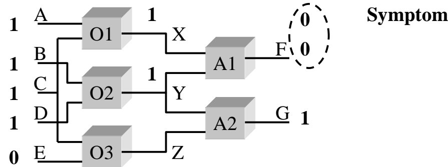
Model-based Diagnosis

- Input: Observations of a system with symptomatic behavior, and a model of the system,
- Output: Diagnoses that account for the symptoms.



Solution: Diagnosis as Hypothesis Testing

1. Test to see if inputs and outputs are consistent.
2. If not, look for mode variables that satisfy the model.



Fault Model: O1's output is stuck to 0 (Output shorted to ground)

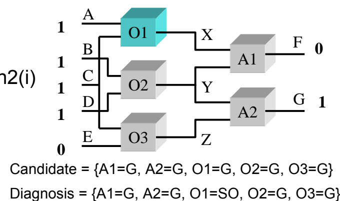
Diagnosis by Generate and Test

- Given: Exhaustive fault models, structure and observations.
- Generate: Consider each mode assignment as a candidate.
- Test:
- 1. Simulate candidate, given inputs.
- 2. Compare to responses
  - Disagree: Discard
  - Agree: Keep
  - No prediction: **Discard**
- Exonerate component if none of its fault models agree
- Problem:
  - Fault models are often incomplete
  - May incorrectly exonerate faulty components

Model-based Diagnosis

Or(i):

- G(i):
  Out(i) = In1(i) or In2(i)
- Stuck_0(i):
  Out(i) = 0



Candidate = {A1=G, A2=G, O1=G, O2=G, O3=G}

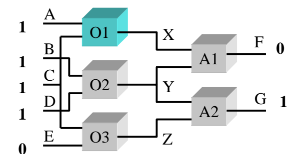
Diagnosis = {A1=G, A2=G, O1=SO, O2=G, O3=G}

- Obs <In, Response>: Assignment to I and R, respectively
- Candidate C_i: Assignment of modes to X
- Diagnosis D_i: A candidate such that $D_i \wedge In \wedge M(X,Y)$ predicts (entails) Response

Model-based Diagnosis

A=1, C=1, O1=G => X=1
B=1, D=1, O2=G => Y=1
C=1, E=0, O3=G => Z=1
X=1, Y=1, A1=G ≠> F = 1
Conclusion: failure

A=1, C=1, O1=SO => X=0
B=1, D=1, O2=G => Y=1
C=1, E=0, O3=G => Z=1
X=0, Y=1, A1=G => F=0
Y=1, Z=1, A2=G => G=1
Conclusion: diagnosis: O1=S0, O2,O3,A1,A2=G



A=1, C=1, O1=G => X=1
B=1, D=1, O2=SO => Y=0
C=1, E=0, O3=G => Z=1
X=1, Y=0, A1=G => F=0
Y=1, Z=1, A2=G => G=1
Conclusion: diagnosis: O2=S0, O1,O3,A1,A2=G

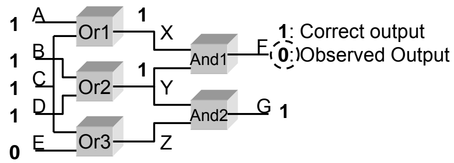
A=1, C=1, O1=G => X=1
B=1, D=1, O2=G => Y=1
C=1, E=0, O3=G => Z=1
X=1, Y=1, A1=S0 => F=0
Y=1, Z=1, A2=G => G=1
Conclusion: diagnosis A1=S0, O1,O2,O3,A2=G

A=1, C=1, O1=G => X=1
B=1, D=1, O2=G => Y=1
C=1, E=0, O3=S0 => Z=0
X=1, Y=1, A1=G => F≠0
Conclusion: O2=S0,... not a diagnosis

A=1, C=1, O1=G => X=1
B=1, D=1, O2=G => Y=1
C=1, E=0, O3=G => Z=1
X=1, Y=1, A1=G => F ≠ 0
Conclusion: A2=S0,… not a diagnosis

How Should Diagnoses Account for Novel Symptoms?

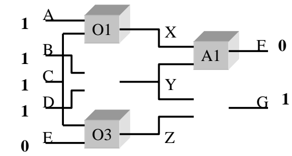
Suspending Constraints: Make no presumption about faulty component behavior.



Consistency-based Diagnosis

And(i):
- G(i):
  Out(i) = In1(i) AND In2(i)
- U(i):

All components have "unknown Mode" U, whose assignment is never mentioned in M



Diagnosis = {A1=G, A2=U O1=G, O2=U, O3=G}

- Obs: Assignment to O
- Candidate C_i: Assignment of modes to X
- Diagnosis D_i: A candidate such that $D_i \wedge Obs \wedge M(X,Y)$ is satisfiable.