Applications of Program analysis in Model-Based Design

Prahlad Sampath (Prahlad.Sampath@mathworks.com)

“©2018 by The MathWorks, Inc., MATLAB, Simulink, Stateflow, are registered trademarks of The MathWorks, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders.”
Outline

- Model-Based Design overview and role of program analysis (20 mins)
- Data flow analysis (20 mins)
- Abstract interpretation (20 mins)
- Software model checking (20 mins)
Model-Based Design Overview
Model-Based Design

- **Design activities**
  - Specification, prototyping (at different levels of fidelity), design space exploration, code-generation, verification

- **Use of appropriate abstractions for modelling – from algorithms to systems**
  - Graphical languages where appropriate
  - Textual languages where appropriate
  - Provide a framework where these can be combined and analysed

- **Analysis is the corner stone of Model-Based Design**
  - Different levels of adoption of Model-Based Design will use different kinds of analysis
Simulink Workflow

Dynamical System

Graphically model the system in Simulink

Simulink Block Diagram

Compilation:
Convert from graphical model to some internal representation

Compiled Information (CI)

\[
X = [x_c, x_d] \\
x_c = f_c(X, U, t) \\
x_d[n + 1] = f_d(X, U, t) \\
y = g(X, U, t) \\
X(t_0) = X_0
\]

Execution Phase

Analyze Results

Present the solution (logging, scopes)

Simulate

Solver Info

\[
t(n + 1) = \ldots
\]

CI

\[
X = [x_c, x_d] \\
x_c = f_c(X, u_1, u_2, t) \\
x_d[n + 1] = g(X, u_1, u_2, t) \\
y = h(X, u_1, u_2, t) \\
X(t_0) = X_0
\]
Modelling Formalisms

- Linear systems (ODEs)

\[
\frac{dv}{dt} = -g, \\
\frac{dx}{dt} = v,
\]

\[v_0 = 15 \text{ m/s} \quad x_0 = 10 \text{ m}\]

\[v^+ = -\kappa v^-, \quad x = 0\]

>> sldemo_bounce_two_integrators
Modelling Formalisms

- Linear systems (ODEs)
Modelling Formalisms

- Linear systems (ODEs)

>>sldemo_bounce_two_integrators
Modelling Formalisms

- Linear systems (ODEs)
- Hierarchical state machines

>>>sldemo_fuelsys
Modelling Formalisms

- Linear systems (ODEs)
- Hierarchical state machines

>>sldemo_fuelsys
Modelling Formalisms

- Linear systems (ODEs)
- Hierarchical state machines
- Tabular specifications

>>ex_stt_boiler
Modelling Formalisms

- Linear systems (ODEs)
- Hierarchical state machines
- Tabular specifications
- Hybrid automata
Modelling Formalisms

- Linear systems (ODEs)
- Hierarchical state machines
- Tabular specifications
- Hybrid automata

>>sf_bounce
Modelling Formalisms

- Linear systems (ODEs)
- Hierarchical state machines
- Tabular specifications
- Hybrid automata
- Differential algebraic system (acausal models for physical systems)
Modelling Formalisms

- Linear systems (ODEs)
- Hierarchical state machines
- Tabular specifications
- Hybrid automata
- Differential algebraic system (acausal models for physical systems)
Modelling Formalisms

- Linear systems (ODEs)
- Hierarchical state machines
- Tabular specifications
- Hybrid automata
- Differential algebraic system (acausal models for physical systems)
- Discrete event systems
- Procedural code
Model-Based Design – Hardware

- Hardware connectivity
- Real time simulation
Model-Based Design – Hardware

- Hardware connectivity
- Real time simulation

With MATLAB support package for Arduino, the Arduino is connected to a computer running MATLAB. Processing is done on the computer with MATLAB.
Model-Based Design – Hardware

- Hardware connectivity
- Real time simulation

With Simulink support package for Arduino, you develop the algorithm in Simulink and deploy to the Arduino using automatic code generation. Processing is then done on the Arduino.
Model-Based Design – Hardware

- Hardware connectivity
- Real time simulation
Model-Based Design – Code generation

- Programming
  - Code generation
  - Design optimizations
  - Verification
Model-Based Design – Code generation

- Programming
  - Code generation
  - Design optimizations
  - Verification
Model-Based Design – Code generation

- Programming
  - Code generation
  - Design optimizations
  - Verification
Model-Based Design – Code generation

- Programming
  - Code generation
  - Design optimizations
  - Verification
Model-Based Design – Code generation

- Programming
  - Code generation
  - Design optimizations
  - Verification
Model-Based Design – Code generation

- Programming
  - Code generation
  - Design optimizations
  - Verification

- Data-type design
Model-Based Design – Code generation

- Programming
  - Code generation
  - Design optimizations
  - Verification

- Data-type design

- Systematic verification
  - Coverage analysis
  - Detect run-time error
  - Property proving
Where does Program Analysis fit into this picture?

- Techniques for reasoning about behaviour
  - Spanning many programming paradigms

- Well developed abstraction techniques
  - Abstraction vs. approximation!

- Scalable algorithms
  - Algorithms that scale to millions of lines of code

- Active area of research
  - New techniques and algorithms
Four properties for a tool

- **Automaticity**
  - Is the tool press-button or does it require user interaction?

- **Soundness**
  - Are results trustable?

- **Completeness**
  - Will the tool find all my bugs?

- **Scalability**
  - How does the tool react to large programs?
Comparison of formal methods properties

- Automaticity
- Soundness
- Completeness
- Scalability

- Abstract Interpretation
- Dynamic test
- Formal proof (Hoare)
- Model Checking
It’s all about the Workflow!
Data Flow Analysis
A Refresher

- Infer “flows” in programs
  - Reason about program statements
  - System of equations that capture the “flow” across statements in a program
  - Algorithms well suited for program graphs
  - Basis for compiler optimizations
A Refresher

- Infer “flows” in programs
  - Reason about program statements
  - System of equations that capture the “flow” across statements in a program
  - Algorithms well suited for program graphs
  - Basis for compiler optimizations
- Generate a set of equations
  - Over a lattice domain
  - Solution calculated by using fixpoint iteration

\[
\begin{align*}
entry(s1) &= exit(e1) \\
& \quad \ldots \\
entry(c1) &= entry(e1) \cup exit(j1) \\
exit(c1) &= entry(c1) - kill(c1) \cup gen(c1) \\
& \quad \ldots \\
entry(j1) &= exit(s3) \cup exit(s3)
\end{align*}
\]
A Refresher

- Infer “flows” in programs
  - Reason about program statements
  - System of equations that capture the “flow” across statements in a program
  - Algorithms well suited for program graphs
  - Basis for compiler optimizations

- Generate a set of equations
  - Over a lattice domain
  - Solution calculated by using fixpoint iteration

- Forward vs. Backward analysis

- Context sensitivity

- Flow sensitivity

- Inter vs. Intra procedural analysis

- …
Applications of data-flow analysis

- Compiler optimizations
  - Expression folding
  - Variable reuse (RAM)
  - Constant folding
  - Strength reduction
Applications of data-flow analysis

- Compiler optimizations
  - Expression folding
  - Variable reuse (RAM)
  - Constant folding
  - Strength reduction

/* Model step function */
void exprfld_step(void)
{
  /* Gain: '<Root>/Gain' incorporates:
   * Inport: '<Root>/In1'
   */
  exprfld_B.S1 = exprfld_P.Gain_Gain * exprfld_U.i1;

  /* Gain: '<Root>/Gain1' incorporates:
   * Inport: '<Root>/In2'
   */
  exprfld_B.S2 = exprfld_P.Gain1_Gain * exprfld_U.i2;

  /* Outport: '<Root>/Out1' incorporates:
   * Product: '<Root>/Product'
   */
  exprfld_Y.Out1 = exprfld_B.S1 * exprfld_B.S2;
}
Applications of data-flow analysis

- Compiler optimizations
  - Expression folding
  - Variable reuse (RAM)
  - Constant folding
  - Strength reduction

```c
/* Model step function */
void exprfld_step(void)
{
    /* Outport: '<Root>/Out1' incorporates:
     * Gain: '<Root>/Gain'
     * Gain: '<Root>/Gain1'
     * Inport: '<Root>/In1'
     * Inport: '<Root>/In2'
     * Product: '<Root>/Product'
     */
    exprfld_Y.Out1 =
        exprfld_P.Gain_Gain *
        exprfld_U.i1 *
        (exprfld_P.Gain1_Gain * exprfld_U.i2);
}
```
Applications of data-flow analysis

- Compiler optimizations
  - Expression folding
  - Variable reuse (RAM)
  - Constant folding
  - Strength reduction

- Parallelization
  - GPU Coder™
Deploy with GPU Coder™

Alexnet

~30 Fps (Tegra X1)

People detection

~66 Fps (Tegra X1)

Vehicle Detection

~20 Fps (K40c)

Lane detection

~130 Fps (K40c)
Debugging Workflow

- Problem:
Debugging Workflow

- Problem:
  - You have a large model, with an error. How should you go about localizing the error?
Debugging Workflow

- Problem:
  - You have a large model, with an error. How should you go about localizing the error?
  - Localize the behaviour over time
    - Simulate the model
    - Insert break-points
    - Visualize signals/outputs
Debugging Workflow

- Problem:
  - You have a large model, with an error. How should you go about localizing the error?
  - Localize the behaviour over time
    - Simulate the model
    - Insert break-points
    - Visualize signals/outputs
  - Localize the behaviour over space
    - Dependency analysis of model
Debugging Workflow

- Problem:
  - You have a large model, with an error. How should you go about localizing the error?
  - Localize the behaviour over time
    - Simulate the model
    - Insert break-points
    - Visualize signals/outputs
  - Localize the behaviour over space
    - Dependency analysis of model
  - Extract and simplify
Debugging Workflow

- **Problem:**
  - You have a large model, with an error. How should you go about localizing the error?
  - Localize the behaviour over time
    - Simulate the model
    - Insert break-points
    - Visualize signals/outputs
  - Localize the behaviour over space
    - Dependency analysis of model
  - Extract and simplify

Google search: “simulink model slicer youtube”
https://www.youtube.com/watch?v=48EAr508mmw
Slicing Models to Reduce Complexity
Abstract Interpretation
Prove that no division by zero occurs here
Prove that no division by zero occurs here

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0; i<=x; i++)
        {
            current_out = in*0.1 + prev_out*0.9;
            prev_out = current_out;
            if (random())
                prev_out = 0.0;
                in = input();
        }
    current_out = current_out + 3.0;
    return 1/current_out;
}
```

Clearly testing is not practicable
- Too many unknown
Prove that no division by zero occurs here

```c
// x is an unknown parameter
float runTheLoop(int x) {
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<x;i++)
        {
            current_out = in*0.1 + prev_out*0.9;
            prev_out = current_out;
            if (random())
                prev_out = 0.0;
                in = input();
        }
    current_out = current_out + 3.0;
    return 1/current_out;
}
```

Clearly testing is not practicable
- Too many unknown

Philosophy of AI to prove correctness:
- You don’t need the exact value of current_out to prove absence of ZDIV
- Having a less precise information e.g. current_out >= 0.2 is sufficient
Formalization of the **concrete semantics of the program**

**States of the program**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (xrandm())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 3.0;
    return 1/current_out;
}
```
Formalization of the **concrete** semantics of the program

**States of the program**

```cpp
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (random())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 3.0;
    return 1/current_out;
}
```

x=1

`current_out=?`

`in=?`

`i=?`
Formalization of the **concrete semantics** of the program

**States of the program**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0; i<=x; i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (i%random())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 5.0;
    return 1/current_out;
}
```

x=1
current_out=0
in=1.6
i=17
Formalization of the **concrete semantics** of the program

**Traces semantics**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (random())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 3.0;
    return 1/current_out;
}
```

Infinite state machine that simulates program execution
Formalization of the **concrete semantics** of the program

**Traces semantics**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        //current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (xrandn())
            prev_out = 0.0;
        in = input();
    } 
    current_out = current_out + 3.0;
    return 1/current_out;
}
```

Infinite state machine that simulates program execution
Formalization of the **concrete semantics** of the program

**Traces semantics**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (x>=x)
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 5.0;
    return 1/current_out;
}
```

Infinite state machine that simulates program execution
Formalization of the **collecting** semantics of the program

*Invariants semantics*

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (isRandom())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 5.0;
    return 1/current_out;
}
```

Iterate until no new states are reached
Formalization of the **collecting** semantics of the program

**Invariants semantics**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (x<max())
            prev_out = 0.0;
    }
    in = input();
}
```

Interate until no new states are reached
Formalization of the **collecting** semantics of the program

**Invariants semantics**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (xrand())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 5.0;
    return 1/current_out;
}
```

Interate until no new states are reached
Formalization of the **collecting** semantics of the program

**Invariants semantics**

```
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (random())
            prev_out = 0.0;
    }  
    in = input();
    current_out = current_out + 3.0;
    return 1/current_out;
}
```

Iterate until no new states are reached
Formalization of the **collecting** semantics of the program

**Invariants semantics**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0; i<=x; i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (random())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 5.0;
    return 1/current_out;
}
```

Interate until no new states are reached
Formalization of the **collecting** semantics of the program

**Invariants semantics**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
  if (x<0 || x>100)
    return 1.0;
  float in = input();
  float current_out = 0.0;
  float prev_out = 0.0;
  int i;
  for (i=0;i<=x;i++)
  {
    current_out = in*0.1 + prev_out*0.9;
    prev_out = current_out;
    if (x<rand())
      prev_out = 0.0;
    in = input();
  }
  current_out = current_out + 5.0;
  return 1/current_out;
}
```

Interate until no new states are reached
Formalization of the **collecting** semantics of the program

**Invariants semantics**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (xenom())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 5.0;
    return 1/current_out;
}
```

Iterate until no new states are reached
Formalization of the **collecting** semantics of the program

**Invariants semantics**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0; i<=x; i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (x<10) {
            prev_out = 0.0;
        }
        in = input();
    }
    current_out = current_out + 5.0;
    return 1/current_out;
}
```

Interate until no new states are reached
Formalization of the **collecting** semantics of the program

**Invariants semantics**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (random())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 5.0;
    return 1/current_out;
}
```

Interate until no new states are reached
Formalization of the collecting semantics of the program

Invariants semantics

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (random())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 3.0;
    return 1.0/current_out;
}

Interate until no new states are reached
```
Formalization of the collecting semantics of the program

**Invariants semantics**

```c
1 // x is an unknown parameter
2 float runTheLoop(int x)
3 {
4   if (x<0 || x>100)
5     return 1.0;
6   float in = input();
7   float current_out = 0.0;
8   float prev_out = 0.0;
9   int i;
10  for (i=0;i<=x;i++)
11    {
12      current_out = in*0.1 + prev_out*0.9;
13      prev_out = current_out;
14      if (random())
15        prev_out = 0.0;
16    in = input();
17    }
18   current_out = current_out + 3.0;
19  return 1/current_out;
20 }
```

Interate until no new states are reached
Formalization of the **collecting** semantics of the program

**Invariants semantics**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (rand()%1)
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 3.0;
    return 1/current_out;
}
```

**Interate until no new states are reached**
Formalization of the collecting semantics of the program

Invariants semantics

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (xrand())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 3.0;
    return 1/current_out;
}
```

Interate until no new states are reached
Formalization of the **abstract** semantics of the program

**Properties semantics – Loss of expressivity**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (random())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 5.0;
    return 1/current_out;
}
```

Finite state machine that **over-approximates** program execution
Formalization of the abstract semantics of the program

Properties semantics – Loss of expressivity

Finite state machine that over-approximates program execution

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0; i<=x; i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (random())
            in = input();
    }
    current_out = current_out + 3.0;
    return 1/current_out;
}
```
Formalization of the **abstract** semantics of the program

**Properties semantics – Loss of expressivity**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
  if (x<0 || x>100)
    return 1.0;
  float in = input();
  float current_out = 0.0;
  float prev_out = 0.0;
  int i;
  for (i=0;i<=x;i++)
  {
    current_out = in*0.1 + prev_out*0.9;
    prev_out = current_out;
    if (random())
      prev_out = 0.0;
    in = input();
  }
  current_out = current_out + 3.0;
  return 1/current_out;
}
```

**Finite state machine that over-approximates program execution**
Formalization of the **abstract** semantics of the program

*Properties semantics – Fixpoint computation*

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (random())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 3.0;
    return 1/current_out;
}
```

Finite state machine that over-approximates program execution
Formalization of the abstract semantics of the program

Properties semantics – Fixpoint computation

Finite state machine that over-approximates program execution
Formalization of the abstract semantics of the program

Properties semantics – Fixpoint computation

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0; i<=x; i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (random())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 3.0;
    return 1/current_out;
}
```

Finite state machine that over-approximates program execution
Formalization of the abstract semantics of the program

Properties semantics – Fixpoint computation

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (x<rand())
            prev_out = 0.0;
    }
    in = input();
    current_out = current_out + 5.0;
    return 1.0/current_out;
}
```

Finite state machine that over-approximates program execution
Formalization of the **abstract** semantics of the program

**Properties semantics – Fixpoint computation**

```c
// x is an unknown parameter
float runTheLoop(int x)
{
  if (x<0 || x>100)
    return 1.0;
  float in = input();
  float current_out = 0.0;
  float prev_out = 0.0;
  int i;
  for (i=0;i<=x;i++)
  {
    current_out = in*0.1 + prev_out*0.3;
    prev_out = current_out;
    if (random())
      prev_out = 0.6;
    in = input();
  }
  current_out = current_out + 3.0;
  return 1/current_out;
}
```

**Finite state machine** that **over-approximates** program execution

Interate until no new states are reached
Formalization of the abstract semantics of the program

Properties semantics – Fixpoint computation

```c
// x is an unknown parameter
float runTheLoop(int x)
{
    if (x<0 || x>100)
        return 1.0;
    float in = input();
    float current_out = 0.0;
    float prev_out = 0.0;
    int i;
    for (i=0;i<=x;i++)
    {
        current_out = in*0.1 + prev_out*0.9;
        prev_out = current_out;
        if (random())
            prev_out = 0.0;
        in = input();
    }
    current_out = current_out + 3.0;
    return 1/current_out;
}
```

Finite state machine that over-approximates program execution

Interate until no new states are reached

Abstract properties represent infinitely many states
Formalization of the abstract semantics of the program

Properties semantics – Fixpoint computation

```
// x is an unknown parameter
float runTheLoop(int x)
{
  if (x<0 || x>100)
    return 1.0;
  float in = input();
  float current_out = 0.0;
  float prev_out = 0.0;
  int i;
  for (i=0; i<=x; i++)
  {
    current_out = in*0.1 + prev_out*0.9;
    prev_out = current_out;
    if (random())
      prev_out = 0.0;
  }  
in = input();
  current_out = current_out + 3.0;
  return current_out;
}
```

Finite state machine that over-approximates program execution

Interate until no new states are reached

Abstract properties represent infinitely many states
Applications of Abstract Interpretation

- Analysis of C Code
  - Code prover
  - Bug finder
Code Prover

- Performs abstract invariants computation on C/C++/ADA programs
  - With pointers
  - With floating points
  - With function calls
  - With multi-tasking
- Zero False Positives
- Computes many properties on program variables
  - Ranges (interval domains) with holes
  - Multiplicity (both integer and floating point)
  - Linear relations between program variables
    \[3x - 2y + 4z \leq 32 \land y + 3z - t \leq 1\]
Simple color code for presenting the information

- **Green:** reliable
  - safe pointer access

- **Red:** faulty
  - out of bounds error

- **Gray:** dead
  - unreachable code

- **Orange:** unproven
  - may be unsafe for some conditions

- **Purple:** violation
  - MISRA-C/C++ or JSF++ code rules

- **Range data**
  - tool tip

```c
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
        *p = 0;
        p++;
    }

    if (get_bus_status() > 0) {
        if (get_oil_pressure() > 0) {
            *p = 5;
        } else {
            i++;
        }
    }
    i = get_bus_status();

    if (i >= 0) {
        if (p - i >= 10) {
            *p - i = 10;
        }
    }
```
Bug Finder

- Bug Finder uses abstract interpretation techniques to find bugs
  - Does not claim to be exhaustive
    \textit{Cannot prove absence of bugs}
  - May find a broader set of bugs
    \textit{Some are hard to deal with in Code Prover}

- Few False Positives
Applications of Abstract Interpretation

- Analysis of C Code
  - Code prover
  - Bug finder
- Analysis of code generated from models
  - Opportunities for using model semantics to improve precision of code-based analysis
Applications of Abstract Interpretation

- Analysis of C Code
  - Code prover
  - Bug finder
- Analysis of code generated from models
  - Opportunities for using model semantics to improve precision of code-based analysis
- Analysis of Models
Applications of Abstract Interpretation

- Analysis of C Code
  - Code prover
  - Bug finder
- Analysis of code generated from models
  - Opportunities for using model semantics to improve precision of code-based analysis
- Analysis of Models
Detailed design – Data-type design
From Specification to Code

- High level design focuses on maximum flexibility in design space exploration
  - Use doubles
  - Let Simulink choose appropriate data-types

- Code generation focuses on designs that are quite precisely specified and optimised
  - Cost of computation
  - Cost of memory
  - Precision requirements
Data type design

- Start with MATLAB algorithm (uses doubles)

```matlab
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end
```
Data type design

- Start with MATLAB algorithm (uses doubles)
- Convert all computations to use fixed-point calculations

```matlab
% Compute Output
if (u_state > limit_upper)
    %F2F: No information found for converting the following block of code
    %F2F: Start block
    y = fi(limit_upper, 0, 16, 7, fm);
    clip_status = fi(-2, 0, 1, 0, fm);
    %F2F: End block
elseif (u_state >= limit_upper)
    %F2F: No information found for converting the following block of code
    %F2F: Start block
    y = fi(limit_upper, 0, 16, 7, fm);
    clip_status = fi(-1, 0, 1, 0, fm);
    %F2F: End block
elseif (u_state < limit_lower)
    %F2F: No information found for converting the following block of code
    %F2F: Start block
    y = fi(limit_lower, 0, 16, 7, fm);
    clip_status = fi(2, 0, 1, 0, fm);
    %F2F: End block
elseif (u_state <= limit_lower)
    %F2F: No information found for converting the following block of code
    %F2F: Start block
    y = fi(limit_lower, 0, 16, 7, fm);
    clip_status = fi(1, 0, 1, 0, fm);
    %F2F: End block
else
    y = fi(u_state, 0, 16, 7, fm);
    clip_status = fi(0, 0, 1, 0, fm);
end
```
Data type design

- Start with MATLAB algorithm (uses doubles)

```matlab
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end
```
Data type design

- Start with MATLAB algorithm (uses doubles)
- Execute test-bench with sample inputs

```matlab
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end
```
Data type design

- Start with MATLAB algorithm (uses doubles)
- Execute test-bench with sample inputs
- Instrument MATLAB to monitor min/max for each variable

```matlab
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end
```
Data type design

- Start with MATLAB algorithm (uses doubles)
- Execute test-bench with sample inputs
- Instrument MATLAB to monitor min/max for each variable
- Propose fixpoint types based on observed ranges

```matlab
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end
```
Data type design

- Start with MATLAB algorithm (uses doubles)
- Execute test-bench with sample inputs
- Instrument MATLAB to monitor min/max for each variable
- Propose fixpoint types based on observed ranges

```
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Sim Min</th>
<th>Sim Max</th>
<th>Whole Num.</th>
<th>Proposed Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>u_in</td>
<td>double</td>
<td>-1</td>
<td></td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>Output</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>double</td>
<td></td>
<td>320.31</td>
<td>No</td>
<td>numerictype(1, 16, 14)</td>
</tr>
<tr>
<td>clip_status</td>
<td>double</td>
<td>0</td>
<td></td>
<td>Yes</td>
<td>numerictype(0, 1, 0)</td>
</tr>
<tr>
<td>Persistent</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u_state</td>
<td>double</td>
<td>2</td>
<td>320.31</td>
<td>No</td>
<td>numerictype(0, 16, 7)</td>
</tr>
<tr>
<td>Local</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>init_val</td>
<td>double</td>
<td>1</td>
<td></td>
<td>Yes</td>
<td>numerictype(0, 1, 0)</td>
</tr>
<tr>
<td>gain_val</td>
<td>double</td>
<td>1</td>
<td></td>
<td>Yes</td>
<td>numerictype(0, 1, 0)</td>
</tr>
<tr>
<td>limit_upper</td>
<td>double</td>
<td>500</td>
<td>500</td>
<td>Yes</td>
<td>numerictype(9, 0)</td>
</tr>
<tr>
<td>limit_lower</td>
<td>double</td>
<td>-500</td>
<td>-500</td>
<td>Yes</td>
<td>numerictype(1, 10, 0)</td>
</tr>
<tr>
<td>tprod</td>
<td>double</td>
<td>-1</td>
<td></td>
<td>No</td>
<td>numerictype(1, 16, 14)</td>
</tr>
</tbody>
</table>
Data type design

- Start with MATLAB algorithm (uses doubles)
- Execute test-bench with sample inputs
- Instrument MATLAB to monitor min/max for each variable
- Propose fixpoint types based on observed ranges
- Change algorithm to use the proposed types
Data type design

- Start with MATLAB algorithm (uses doubles)
- Execute test-bench with sample inputs
- Instrument MATLAB to monitor min/max for each variable
- Propose fixpoint types based on observed ranges
- Change algorithm to use the proposed types
- Verify behaviour on test-bench
Data type design

- Start with MATLAB algorithm (uses doubles)

```matlab
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end
```
Data type design

- Start with MATLAB algorithm (uses doubles)
- Propose fixed-point types based on Abstract Interpretation

```matlab
% Compute Output
if (u_state > limit_upper)
    y = limit_upper;
    clip_status = -2;
elseif (u_state >= limit_upper)
    y = limit_upper;
    clip_status = -1;
elseif (u_state < limit_lower)
    y = limit_lower;
    clip_status = 2;
elseif (u_state <= limit_lower)
    y = limit_lower;
    clip_status = 1;
else
    y = u_state;
    clip_status = 0;
end
```
Data type design

- Start with MATLAB algorithm (uses doubles)
- Propose fixed-point types based on Abstract Interpretation
Data type design

- Start with MATLAB algorithm (uses doubles)
- Propose fixed-point types based on Abstract Interpretation
- Change algorithm to use the proposed types
Data type design

- Start with MATLAB algorithm (uses doubles)
- Propose fixed-point types based on Abstract Interpretation
- Change algorithm to use the proposed types
- Verify behaviour on test-bench
Software Model Checking
Model Checking

- Start with a model
  - Various forms of state machines/transition systems
  - Represents a system

- Add a property
  - Various forms of logic – temporal, modal etc.
  - Represents a property

- Algorithms to check the satisfiability/validity of the property on the model
Software Model Checking

- Model is software
  - Complex control-flow
  - Functions/procedures
  - Data-structures, pointers
  - Libraries
  - ...

...
Software Model Checking

- Model is software
  - Complex control-flow
  - Functions/procedures
  - Data-structures, pointers
  - Libraries
  - ...

- Some properties can also be encoded in the software
  - ASSERT statements
Software Model Checking

- Model is software
  - Complex control-flow
  - Functions/procedures
  - Data-structures, pointers
  - Libraries
  - ...

- Some properties can also be encoded in the software
  - ASSERT statements

- Prove that the ASSERT statements cannot be hit or provide evidence of reachability
Model-Based Software Model Checking

- Model is the software (is the model)!
Model-Based Software Model Checking

- Model is the software (is the model)!
  - Discrete state Simulink diagrams
Model-Based Software Model Checking

- Model is the software (is the model)!
  - Discrete state Simulink diagrams
  - Stateflow
Model-Based Software Model Checking

- Model is the software (is the model!)
  - Discrete state Simulink diagrams
  - Stateflow
  - Tabular notations
Model-Based Software Model Checking

- Model is the software (is the model):
  - Discrete state Simulink diagrams
  - Stateflow
  - Tabular notations
  - Integrators, Lookup tables,
Model-Based Software Model Checking

- Model is the software (is the model)!
  - Discrete state Simulink diagrams
  - Stateflow
  - Tabular notations
  - Integrators, Lookup tables,
  - Signals, Triggers, Function calls, Subsystems, Reference Models
Model-Based Software Model Checking

- Model is the software (is the model!)
  - Discrete state Simulink diagrams
  - Stateflow
  - Tabular notations
  - Integrators, Lookup tables,
  - Signals, Triggers, Function calls, Subsystems, Reference Models
  - Arrays of signals, Buses (structures) of signals
Model-Based Software Model Checking

- Model is the software (is the model!)
  - Discrete state Simulink diagrams
  - Stateflow
  - Tabular notations
  - Integrators, Lookup tables,
  - Signals, Triggers, Function calls, Subsystems, Reference Models
  - Arrays of signals, Buses (structures) of signals
  - MATLAB code
Model-Based Software Model Checking

- Model is the software (is the model!)
Model-Based Software Model Checking

- Model is the software (is the model!)
- Property is also modelled
  - Assertion blocks
  - Assumption blocks
Model-Based Software Model Checking

- Model is the software (is the model!)
- Property is also modelled
  - Assertion blocks
  - Assumption blocks
Model-Based Software Model Checking

- Model is the software (is the model!)
- Property is also modelled
  - Assertion blocks
  - Assumption blocks
  - Special statements in MATLAB code
Model-Based Software Model Checking

- Model is the software (is the model!)
- Property is also modelled
  - Assertion blocks
  - Assumption blocks
  - Special statements in MATLAB code
Model-Based Software Model Checking

- Model is the software (is the model!)
- Property is also modelled
  - Assertion blocks
  - Assumption blocks
  - Special statements in MATLAB code
- Prove properties valid
Model-Based Software Model Checking

- Model is the software (is the model!)
- Property is also modelled
  - Assertion blocks
  - Assumption blocks
  - Special statements in MATLAB code
- Prove properties valid
- Provide evidence of invalidity
Run-time Error Check

- Integer overflow
- Division by zero
- Array out-of-bounds
- Range violations
- Dead Logic
Test Generation Analysis

- Time varying inputs to satisfy coverage conditions
- Generate tests for missing coverage
- Special heuristics to satisfy timers/counters
Property Proving

- Functional verification and witness generation for safety properties
- Check for bounded depth validation or for full guarantees

Design augmented with modeled requirements

Highlighted Model

Counter-example
It’s all about the Workflow!
Opportunities at MathWorks Bangalore

- Teams working on multiple areas of MATLAB and Simulink
- IN-SLVnV team
  - Simulink Design Verifier
  - Simulink Solvers and Execution Engine
  - HDL Code Generation and optimizations
  - Simulink Model Advisor
  - Simulink Test
  - Simulink Coder and optimizations
  - MATLAB Coder and optimizations
- Masters with some industry experience or PhD
  - Passion for developing software