Formal Verification of A Domain Specific Language for Run-time Adaptation

Shahid Khan [1] Faiq Khalid [2] Osman Hasan [1] João M.P. Cardoso [3]

[1]School of Electrical Engineering & Computer Science
National University of Sciences & Technology, Islamabad, Pakistan

[2]Department of Computer Engineering
Vienna University of Technology, Vienna, Austria

[3]Faculty of Engineering, University of Porto, Porto, Portugal

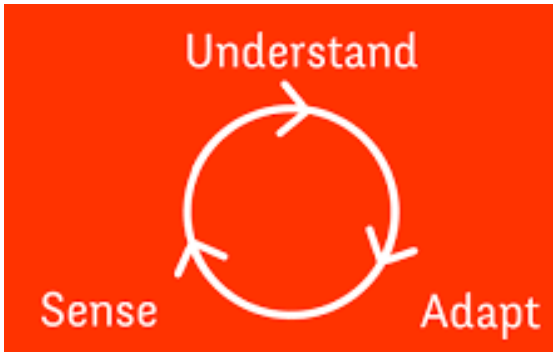
SysCon-2018
Vancouver, Canada

April 24, 2018

Outline

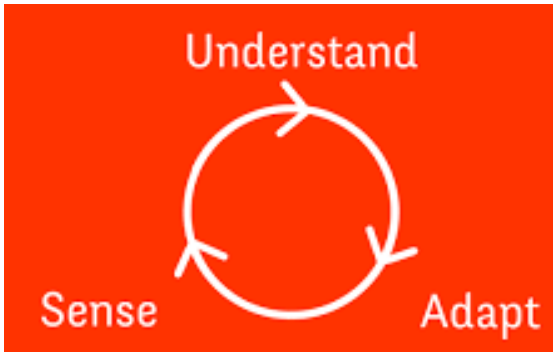
Run-Time Adaptations

- Most of the embedded/IoTs based systems have very high performance requirements, which vary with changing environments



Run-Time Adaptations

- Most of the embedded/IoTs based systems have very high performance requirements, which vary with changing environments



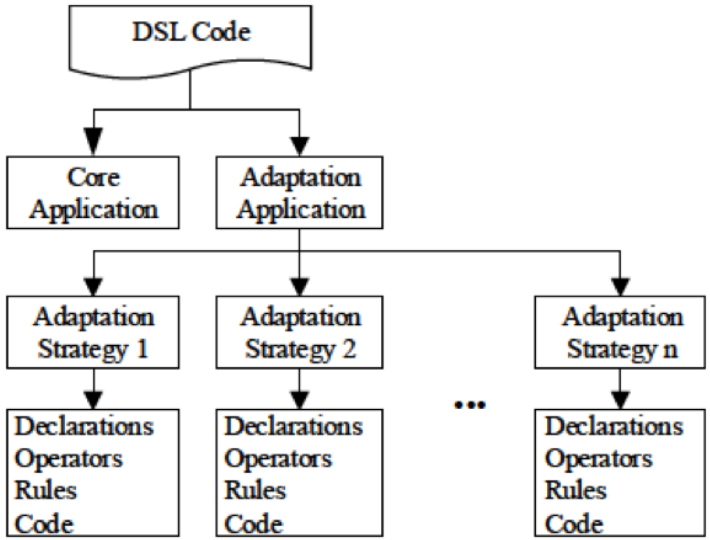
- Run-time software adaptations allow us to update both functional and nonfunctional behavior of the software at runtime

Run-Time Adaptations

- General-purpose programming languages, like C/C++/Java, do not support programming of dynamic adaptive behaviors

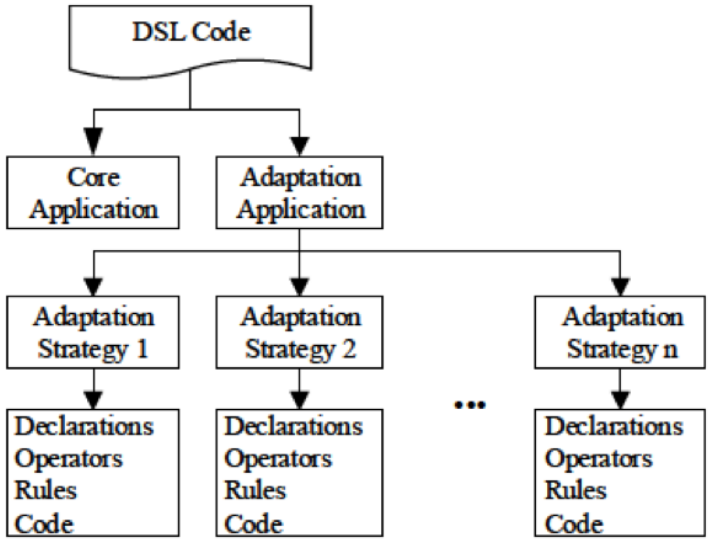
Run-Time Adaptations

- General-purpose programming languages, like C/C++/Java, do not support programming of dynamic adaptive behaviors
- Domain specific languages (DSLs) cater for this by partitioning software development into static and dynamic parts



Run-Time Adaptations

- General-purpose programming languages, like C/C++/Java, do not support programming of dynamic adaptive behaviors
- Domain specific languages (DSLs) cater for this by partitioning software development into static and dynamic parts



- Rules govern the adaptation behavior

Rule Conflict Analysis

- Incorrect specification of rules may lead to Reachability Issues or Conflicts or even functional bugs
  - Several rules can be defined in a single adaptation strategy
  - Rules can run sequentially or concurrently
- Example

```
1: infAct IS fftKnnInf(FftNrSamples=2048)
2:  { IDLE, WALK, JOG };
3:  RUN[period=1sec]
4:   {
5:    //default implementation
6:     activityContext = infAct();
7:   }
8:  RULES
9:   {
10:      EVENT: ENERGY.LEVEL.LOW || CPU.LOAD.HIGH
11:       { //Rule A
12:        infAct IS fftKnnInf(FftNrSamples=512);
13:       }
14:      EVERY RUN: RUN.ELAPSEDTIME > RUN.period
15:       { // Rule B
16:       }
17:  }
```

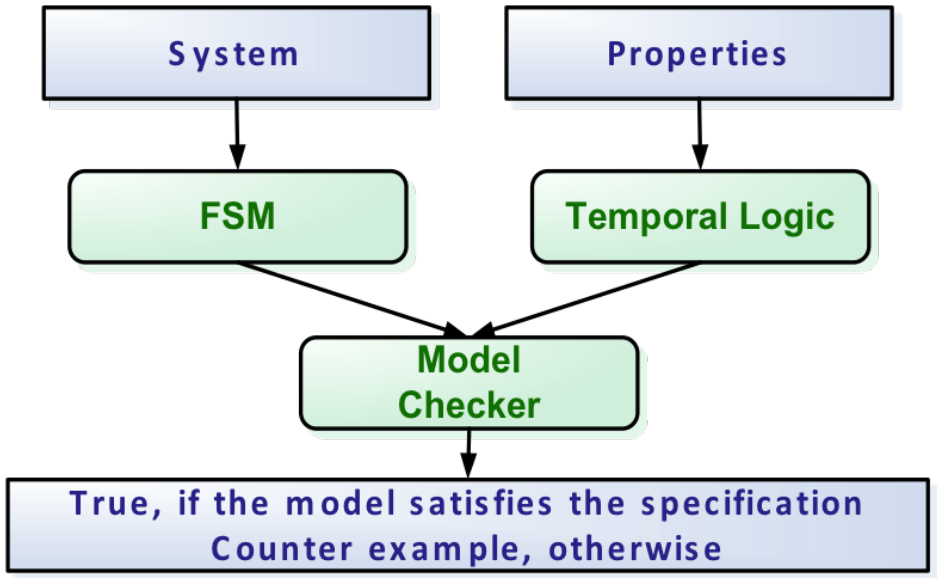
Rule Conflict Analysis Approaches

- Manual manipulation of automata based models
  - Human Error Prone
  - May miss conflicts

Rule Conflict Analysis Approaches

- Manual manipulation of automata based models
  - Human Error Prone
  - May miss conflicts
- Software testing
  - In-exhaustive
  - May miss conflicts

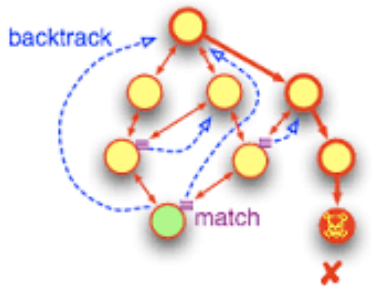
Model Checking

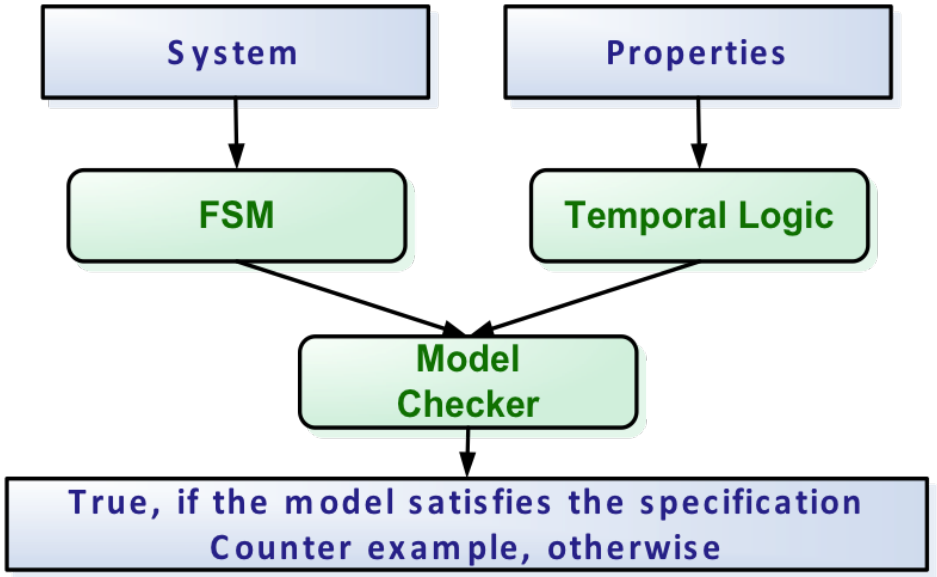


Model Checking

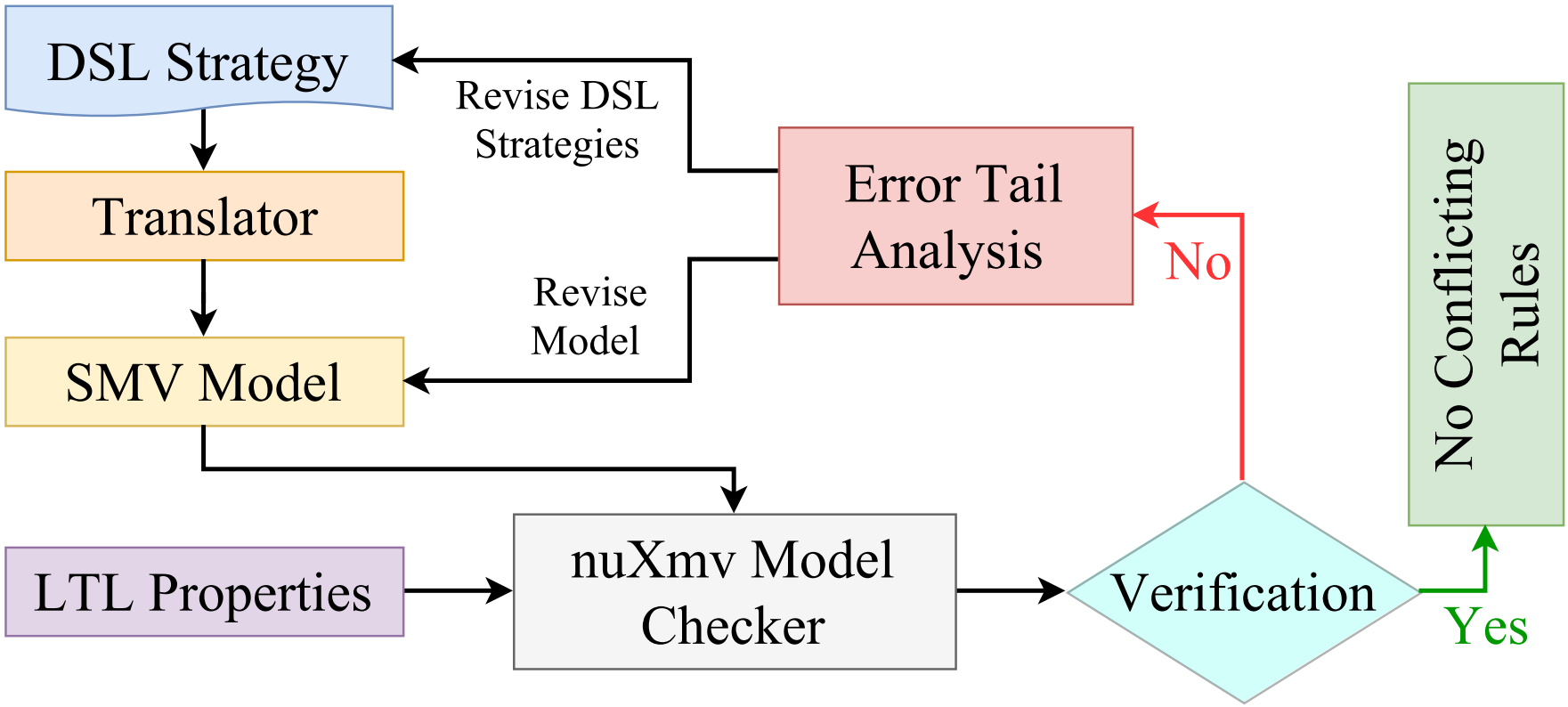
model checking:

all program state are explored
until none left or defect found





Proposed Methodology



Example: Formal Model of a DSL Strategy

Algorithm 1 Pseudocode of the SMV Model

```
1: MODULE main
Require: 0 ≥ EnergyLevel ≤ 100 and Memory ≥ 0
2: Instantiate Modules
3: re : energy(EnergyLevel)                    rules
4: rm : memory(Memory)                         {
5: MODULE energy(EnergyLevel)                      r_energy : every (1 sec)
Ensure: energyBudget                               {
6: if EnergyLevel < 10 then                         if( energy <10)
7:     energyBudget = 50                            { energyBudget = 50
8: else                                                 //...
9:     energyBudget = energyBudget                  }
10: end if                                          }
11: MODULE memory(Memory)                           r_memory : every (1 sec)
Ensure: energyBudget                               {
12: if Memory < 1000 then                           if( memory <1000)
13:     energyBudget = 75                           { energyBudget = 75
14: else                                                //...
15:     energyBudget = energyBudget                 }
16: end if                                         }}
```

Example: Formal Model of a DSL Strategy

Algorithm 1 Pseudocode of the SMV Model

```
1: MODULE main
Require: 0 ≥ EnergyLevel ≤ 100 and Memory ≥ 0
2: Instantiate Modules
3: re : energy(EnergyLevel)                    rules
4: rm : memory(Memory)                         {
5: MODULE energy(EnergyLevel)                    r_energy : every (1 sec)
Ensure: energyBudget                             {
6: if EnergyLevel < 10 then                       if( energy <10)
7:     energyBudget = 50                          { energyBudget = 50
8: else                                             //...
9:     energyBudget = energyBudget                 }
10: end if                                         }
11: MODULE memory(Memory)                        r_memory : every (1 sec)
Ensure: energyBudget                             {
12: if Memory < 1000 then                         if( memory <1000)
13:     energyBudget = 75                         { energyBudget = 75
14: else                                            //...
15:     energyBudget = energyBudget                }
16: end if                                        }}
```

- Values are assigned to the variables Non-deterministically

Formal Verification

- Identified 5 Properties
  - We believe that the successful verification of these properties would ensure that there are no functional bugs, including conflicts, in the given strategy

Property 1: Reachability

Definition

Every state of the model has at least one incoming and one outgoing transition

Property 1: Reachability

Definition

Every state of the model has at least one incoming and one outgoing transition

- Example:
  - Adjusts the number of samples considered by a Fast Fourier Transform (FFT) algorithm from 2,048 to 512 whenever CPU load increases from 80%

Example

```
G (CPU_LOAD < 80 & FFT.FftNrSamples = 512)
     -> F (FFT.FftNrSamples = 2048))
```

Property 2: Trigger Similarity

Definition

Multiple triggering conditions occurring simultaneously do not lead to contradictory behavior

Property 2: Trigger Similarity

Definition

Multiple triggering conditions occurring simultaneously do not lead to contradictory behavior

- Example:
  - Adaptation1 and Adaptation2 are contradictory

Example

```
G !(Adaptation1 & Adaptation2)
```

Property 3: Action Similarity

Definition

Concurrent or sequential rules do not access the same resources or parameters

Property 3: Action Similarity

Definition

Concurrent or sequential rules do not access the same resources or parameters

- Example:
  - Two concurrent rules memory and energy should not set a single parameter energyBudget with different values

Example

```
G (r1.energyBudget = r2.energyBudget)
```

Property 4: Overriding Rules

Definition

No rules should become redundant under any triggering conditions

Property 4: Overriding Rules

Definition

No rules should become redundant under any triggering conditions

- Example:
  - Three energy rules that set energy state of the system to A , B , C based upon whether the energy level is less than 10, 20 or 30, respectively

Example

```
G ( EnergyLevel < 10 -> X state = A)
G ( EnergyLevel < 20 -> X state = B)
G ( EnergyLevel < 30 -> X state = C)
```

Property 5: Incompatible Requirements

Definition

Ensure that different adaptation requirements do not contradict each other

Property 5: Incompatible Requirements

Definition

Ensure that different adaptation requirements do not contradict each other

- Example:
  - Rule r1 moves the system to a low energy state and another rule r2 tries to increase its computation rate

Example

```
G !(r1.state=LowEnergy & r2.state=HighRate)
```

Outline

PAST Algorithm

- Energy aware CPU clock setting algorithm

Initialize:
run_cycles: Number of non-idle CPU Cycles in last interval
$idle\_cycles$: Number of Idle CPU Cycles
$excess\_cycles$: Number of Cycles left over from previous interval

1: **while** (1) **do**
2: $idle\_cycles = hard\_idle + soft\_idle$
3: $run\_cycles+ = excess\_cycles$
4: $run\_percent = run\_cycles/(idle\_cycles + run\_cycles)$
5: $next\_excess = run\_cyclesspeed * (run\_cycles + soft\_idle)$
6: **if** $excess\_cycles < 0$ **then**
7: $excess\_cycles = 0$
8: **end if**
9: $energy = (run\_cycles - excess\_cycles) * speed * speed$
10: **if** $excess\_cycles > idle\_cycles$ **then**
11: $newspeed = 1.0$
12: **else if** $run\_percent > 0.7$ **then**
13: $newspeed = speed + 0.2$
14: **else if** $run\_percent < 0.5$ **then**
15: $newspeed = speed - (0.6 - run\_percent)$
16: **end if**
17: **if** $newspeed > 1.0$ **then**
18: $newspeed = 1.0$
19: **end if**
20: **if** $newspeed < min\_speed$ **then**
21: $newspeed = min\_speed$
22: **end if**
23: $speed = newspeed$
24: $excess\_cycles = next\_excess$
25: **end while**

PAST Algorithm

- Energy aware CPU clock setting algorithm

Initialize:
run_cycles: Number of non-idle CPU Cycles in last interval
$idle\_cycles$: Number of Idle CPU Cycles
$excess\_cycles$: Number of Cycles left over from previous interval

1: **while** (1) **do**
2: $idle\_cycles = hard\_idle + soft\_idle$
3: $run\_cycles+ = excess\_cycles$
4: $run\_percent = run\_cycles/(idle\_cycles + run\_cycles)$
5: $next\_excess = run\_cyclesspeed * (run\_cycles + soft\_idle)$
6: **if** $excess\_cycles < 0$ **then**
7: $excess\_cycles = 0$
8: **end if**
9: $energy = (run\_cycles - excess\_cycles) * speed * speed$
10: **if** $excess\_cycles > idle\_cycles$ **then**
11: $newspeed = 1.0$
12: **else if** $run\_percent > 0.7$ **then**
13: $newspeed = speed + 0.2$
14: **else if** $run\_percent < 0.5$ **then**
15: $newspeed = speed - (0.6 - run\_percent)$
16: **end if**
17: **if** $newspeed > 1.0$ **then**
18: $newspeed = 1.0$
19: **end if**
20: **if** $newspeed < min\_speed$ **then**
21: $newspeed = min\_speed$
22: **end if**
23: $speed = newspeed$
24: $excess\_cycles = next\_excess$
25: **end while**

```
1:  MODULE main
2:   VAR
3:    excess_cycles : real;
4:    next_excess : real;
5:    run_cycles : real;
6:    newspeed : real;
7:   DEFINE
8:    hard_idle := 10;
9:    soft_idle := 0;
10:   idle_cycles := hard_idle + soft_idle;
11:   run_percent := run_cycles/idle_cycles * ( 1 - run_cycles/idle_cycles +
                     ((run_cycles/idle_cycles)^2 - (run_cycles/idle_cycles)^3);
12:   speed := newspeed;
13:   energy := (run_cycles - excess_cycles) *
                  speed^2;
14:   min_speed := 0.2;
15:   ASSIGN
16:    init(run_cycles):= 0;
17:    init(run_cycles):= run_cycles +
                          next(excess_cycles);
18:    init(newspeed):= min_speed;
19:    next(newspeed):=
20:     case
21:      excess_cycles > idle_cycles : 1;
22:      run_percent > 0.7 : speed + 0.2;
23:      run_percent < 0.5 : speed - (0.6 -
                                      run_percent);
24:      newspeed > 1 : 1;
25:      newspeed < min_speed : min_speed;
26:      TRUE : min_speed;
27:     esac;
28:    init(next_excess):= run_cycles - speed *
                          (run_cycles + soft_idle);
29:    next(next_excess):= run_cycles - speed *
                          (run_cycles + soft_idle);
30:    init(excess_cycles):= next_excess;
31:    next(excess_cycles):=
32:     case
33:       excess_cycles < 0 : 0;
34:       TRUE : next_excess;
35:     esac;
```

PAST Algorithm

Two strategies

- Speed Bound
  - Selects the clock speed in such a way that it remains within the given bound
  - Only one rule so no conflicts found

PAST Algorithm

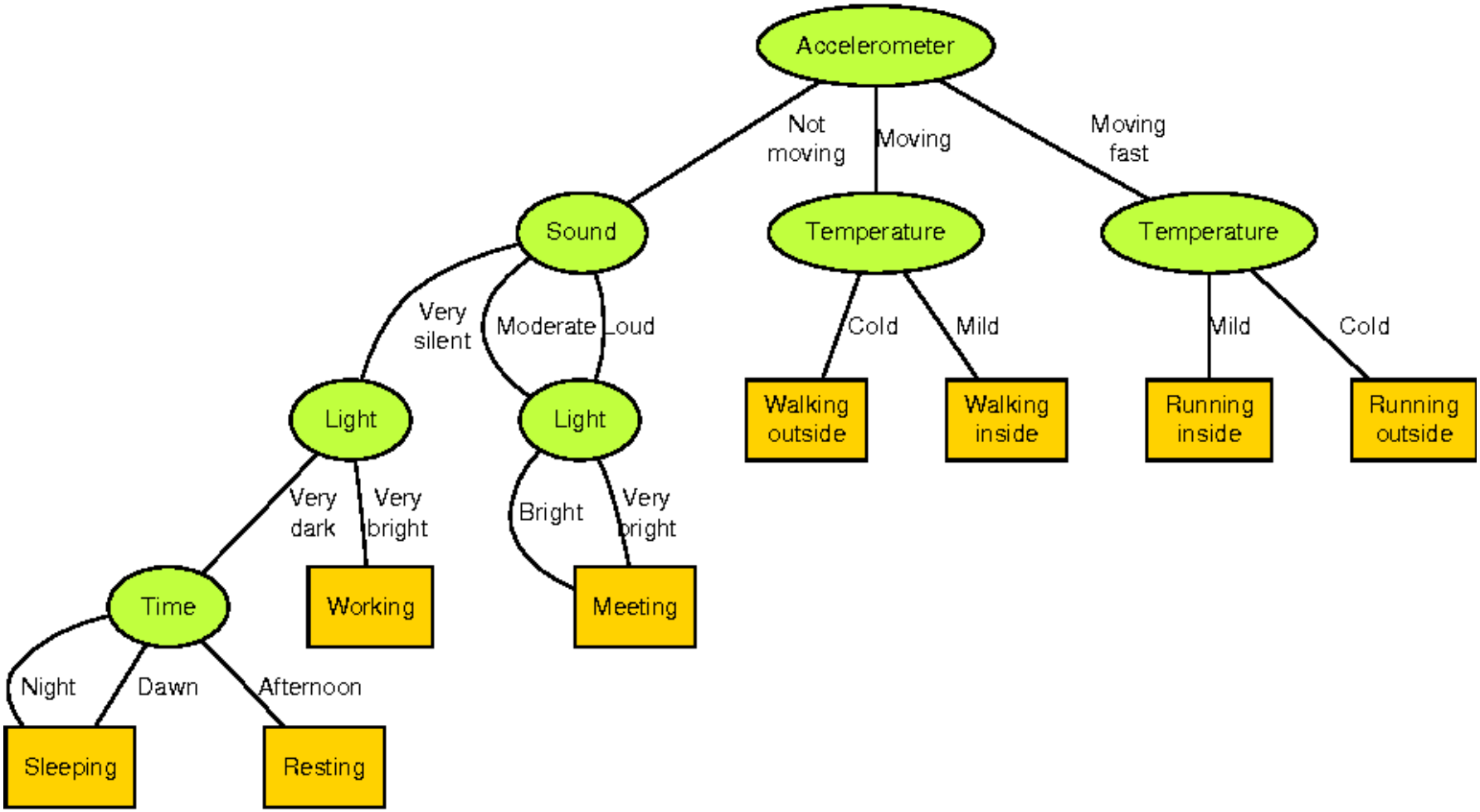
Two strategies

- Speed Bound
  - Selects the clock speed in such a way that it remains within the given bound
  - Only one rule so no conflicts found
- Energy Optimization
  - CPU operates at the minimum clock speed during the idle time
  - Failed the reachability property
    - PAST algorithm does not select the clock speed when the run percent lies between 0.5 and 0.7, which means that the CPU does not operate if it remains 50% to 70% active

Outline

Context Aware Application

Human activity monitoring and then adjusts its behavior accordingly



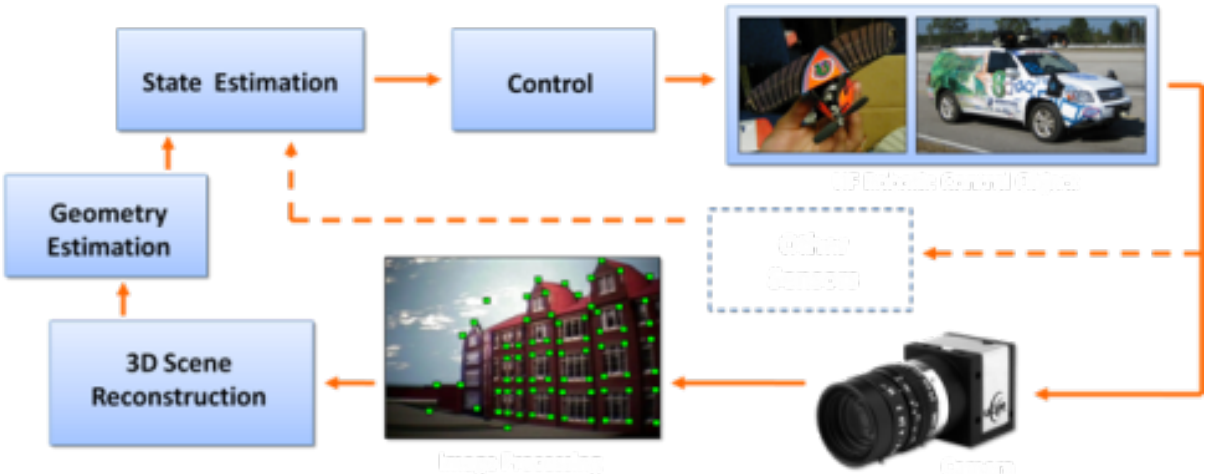
Context Aware Application

Strategy Name	Number of Rules	Failing Property	Failing Rules	Time
Strategy 2	2	Reachability	2	0.06
		Action Similarity	2	0.03
Strategy 3	4	Reachability	4	0.05
		Reachability	2	0.02
		Reachability	2	0.02
		Reachability	2	0.11
		Overriding Rules	2	0.02
Strategy 4	2	Reachability	1	0.03
		Reachability	1	0.03
		Reachability	1	0.05
		Incompatible Requirement	2	0.05
Strategy 5	3	Reachability	1	0.03
		Reachability	1	0.02
		Incompatible Requirements	2	0.03
		Action Similarity	2	0.03
Strategy 6	4	Reachability	1	0.02
		Reachability	1	0.03
		Action Similarity	2	0.02
		Action Similarity	2	0.06
		Overriding Rules	2	0.05

Outline

Stereo Navigation

Can be used to perform the navigation related activities with the help of a camera and a map



Strategy Name	Number of Rules	Failing Property	Failing Rules	Time
Strategy 6	3	Action Similarity	2	0.52
		Action Similarity	2	0.38

Conclusion

- Summary
  - A symbolic model checking based methodology for formal rule conflict and reachability analysis of a domain specific language (DSL)
  - Identification of formal properties for detecting rule conflicts and reachability problems
  - Bugs identified in 3 real-world case studies that were taken from the PhD thesis that proposed DSLs for runtime adaptations

Conclusion

- Summary
  - A symbolic model checking based methodology for formal rule conflict and reachability analysis of a domain specific language (DSL)
  - Identification of formal properties for detecting rule conflicts and reachability problems
  - Bugs identified in 3 real-world case studies that were taken from the PhD thesis that proposed DSLs for runtime adaptations
- Future Work
  - Analyzing other embedded domain languages, e.g., LAnguage for Reconfigurable Architectures (LARA)

Thanks!

save.seecs.nust.edu.pk