Scalable Software Testing and Verification Through Heuristic Search and Optimization: Experiences and Lessons Learned

Lionel Briand

Interdisciplinary Centre for ICT Security, Reliability, and Trust (SnT) University of Luxembourg, Luxembourg

GDR-GPL, Bordeaux, June 2015
Acknowledgements

PhD. Students:
• Vahid Garousi
• Marwa Shousha
• Zohaib Iqbal
• Reza Matinnejad
• Stefano Di Alesio

Scientists:
• Shiva Nejati
• Andrea Arcuri
• Yvan Labiche
• Arnaud Gotlieb
Scalable Software Testing and Verification Through Heuristic Search and Optimization
The term “verification” is used in its wider sense: Defect detection.

Testing is, in practice, the most common verification technique.

Testing is about systematically, and preferably automatically, exercise a system such as to maximize chances of uncovering (important) latent faults within time constraints.

Other forms of verifications are important too (e.g., design time, run-time), but much less present in practice.

Decades of research have not yet significantly and widely impacted software verification practice.
Scalable? Applicable?

- **Scalable**: Can a technology be applied on large artifacts (e.g., models, data sets, input spaces) and still provide useful support within reasonable effort, CPU and memory resources?

- **Applicable**: Can a technology be efficiently and effectively applied by engineers in realistic conditions?
  - realistic ≠ universal
  - includes usability
Focus

• *Formal Verification (Wikipedia):* In the context of hardware and software systems, formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.

• *Our focus:* How can we, in a practical, effective and efficient manner, uncover as many (critical) faults as possible in software systems, within time constraints, while scaling to artifacts of realistic size.
Metaheuristics

• “A **metaheuristic** is a heuristic method for solving a very general class of computational problems by combining user given black-box procedures — usually heuristics themselves — in a hopefully efficient way.” (Wikipedia)

• Hill climbing, Tabu search, Simulated Annealing, Genetic algorithms, Ant colony optimisation ….

• Our research is agnostic to any specific technology but is driven by problems – the use of metaheuristics is however a recurring pattern. Why?
Talk Outline

• Context

• Selected project examples, with industry collaborations

• Similarities and patterns

• Lessons learned
Context
SnT Software Verification and Validation Lab

- SnT centre, Est. 2009: Interdisciplinary, ICT security-reliability-trust
- 230 scientists and Ph.D. candidates, 20 industry partners
- 25 scientists (Research scientists, associates, and PhD candidates)
- Industry-relevant research on system dependability: security, safety, reliability
- Partners: Cetrel, CTIE, Delphi, SES, IEE, Hitec …
Collaborative Model of Research and Innovation

Schneiderman, 2013

- Basic and applied research take place in a rich context
- Basic Research is also driven by problems raised by applied research, which is itself fed by innovation and development
- Publishable research results and focused practical solutions that serve an existing market.
Collaboration in Practice

- Well-defined problems in context
- Realistic evaluation
- Long term industrial collaborations
Testing Software Controllers

References:

• R. Matinnejad et al., “Effective Test Suites for Mixed Discrete-Continuous Stateflow Controllers”, ACM ESEC/FSE 2015
Electronic Control Units (ECUs)

Comfort and variety

More functions
Safety and reliability

Faster time-to-market
Greenhouse gas emission laws

Less fuel consumption
Dynamic Continuous Controllers
Different testing strategies are required for different types of functions.
Development Process

Model-in-the-Loop Stage
- Simulink Modeling
- MiL Testing

Software-in-the-Loop Stage
- Code Generation and Integration
- SiL Testing

Hardware-in-the-Loop Stage
- Software Running on ECU
- HiL Testing
MATLAB/Simulink model

Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, …
Controller Input and Output at MIL

![Diagram of controller input and output]

- **Desired value**
- **Error**
- **Controller (SUT)**
- **Plant Model**
- **System output**

**Test Input**
- Initial Desired Value
- Final Desired Value

**Test Output**
- Desired Value
- Actual Value

Time:
- $T/2$
- $T$
Controllers at MIL

Inputs: Time-dependent variables

Plant Model

actual(t) - desired(t)

output(t)

Configuration Parameters

Inputs: Time-dependent variables

$K_P e(t)$

$K_I \int e(t) \, dt$

$K_D \frac{de(t)}{dt}$

P

I

D
Requirements and Test Objectives

- Initial Desired (ID)
- Desired Value (input)
- Actual Value (output)
- Final Desired (FD)
- Smoothness
- Responsiveness
- Stability
Test Strategy: A Search-Based Approach

- Continuous behavior
- Controller’s behavior can be complex
- Meta-heuristics in (large) input space: Finding worst case inputs
- Possible because of automated oracle (feedback loop)
- Different worst cases for different requirements
- Worst cases may or may not violate requirements
Smoothness Objective Functions: $O_{\text{Smoothness}}$

Test Case A

Test Case B

$O_{\text{Smoothness}}$(Test Case A) $> O_{\text{Smoothness}}$(Test Case B)

We want to find test scenarios which maximize $O_{\text{Smoothness}}$
Search Elements

• **Search Space:**
  • Initial and desired values, configuration parameters

• **Search Technique:**
  • (1+1) EA, variants of hill climbing, GAs …

• **Search Objective:**
  • Objective/fitness function for each requirement

• **Evaluation of Solutions**
  • Simulation of Simulink model => fitness computation

• **Result:**
  • Worst case scenarios or values to the input variables that (are more likely to) break the requirement at MiL level
  • Stress test cases based on actual hardware (HiL)
Solution Overview (Simplified Version)

Objective Functions based on Requirements + Controller-plant model

1. Exploration

HeatMap Diagram

Domain Expert

List of Critical Regions

2. Single-State Search

Worst-Case Scenarios

Graph Builder

Final vs. Initial

Smoothness

Initial Desired

Final Desired

Desired Value

Actual Value
Automotive Example

- **Supercharger bypass flap controller**
  - Flap position is bounded within [0..1]
  - Implemented in MATLAB/Simulink
  - 34 sub-components decomposed into 6 abstraction levels
  - The simulation time $T = 2$ seconds

Flap position = 0 (open)  
Flap position = 1 (closed)
Finding Seeded Faults

Inject Fault

Figure 1
Analysis – Fitness increase over iterations
Analysis II – Search over different regions

![Graphs showing average, (1+1) EA distribution, and Random Search distribution over number of iterations.](image)
Conclusions

• We found much worse scenarios during MiL testing than our partner had found so far, and much worse than random search (baseline)
• These scenarios are also run at the HiL level, where testing is much more expensive: MiL results -> test selection for HiL
• But further research was needed:
  – Simulations are expensive
  – Configuration parameters (ASE 2014)
  – Dynamically adjust search algorithms in different subregions (exploratory <-> exploitative)
Testing in the Configuration Space

- MIL testing for all feasible configurations
- The search space is much larger
- The search is much slower (Simulations of Simulink models are expensive)
- Results are harder to visualize
- Not all configuration parameters matter for all objective functions
Modified Process and Technology

1. Exploration with Dimensionality Reduction
   - Regression Tree
   - Domain Expert
   - List of Critical Partitions

2. Search with Surrogate Modeling
   - Visualization of the 8-dimension space using regression trees
   - Surrogate modeling to predict the objective function and speed up the search

Objective Functions + Controller Model (Simulink)
Dimensionality Reduction

• Sensitivity Analysis: Elementary Effect Analysis (EEA)
• Identify non-influential inputs in computationally costly mathematical models
• Requires less data points than other techniques
• Observations are simulations generated during the Exploration step
• Compute sample mean and standard deviation for each dimension of the distribution of elementary effects
Imagine function F with 2 inputs, x and y:

Elementary Effects

for X

F(A1)-F(A)
F(B1)-F(B)
F(C1)-F(C)
...

for Y

F(A2)-F(A)
F(B2)-F(B)
F(C2)-F(C)
...
Surrogate Modeling (1)

• Goal: To predict the value of the objective functions within a critical partition, given a number of observations, and use that to avoid as many simulations as possible and speed up the search.
Surrogate Modeling (2)

- Any supervised learning or statistical technique providing fitness predictions with confidence intervals

1. Predict higher fitness with high confidence: Move to new position, no simulation
2. Predict lower fitness with high confidence: Do not move to new position, no simulation
3. Low confidence in prediction: Simulation
The best regression technique to build Surrogate models for all of our three objective functions is Polynomial Regression with $n = 3$.

Other supervised learning techniques, such as SVM.

Mean of $R^2$/MRPE values for different surrogate modeling techniques

<table>
<thead>
<tr>
<th></th>
<th>LR $R^2$/MRPE</th>
<th>ER $R^2$/MRPE</th>
<th>PR($n=2$) $R^2$/MRPE</th>
<th>PR($n=3$) $R^2$/MRPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_{sm}$</td>
<td>0.66/0.0526</td>
<td>0.44/0.0791</td>
<td>0.95/0.0203</td>
<td>0.98/0.0129</td>
</tr>
<tr>
<td>$F_r$</td>
<td>0.78/0.0295</td>
<td>0.49/1.2281</td>
<td>0.85/0.0247</td>
<td>0.85/0.0245</td>
</tr>
<tr>
<td>$F_{st}$</td>
<td>0.26/0.2043</td>
<td>0.22/1.2519</td>
<td>0.46/0.1755</td>
<td>0.54/0.1671</td>
</tr>
</tbody>
</table>
Experiments Results (RQ2)

✓ Dimensionality reduction helps generate better surrogate models for Smoothness and Responsiveness requirements.

Mean Relative Prediction Errors (MRPE Values)

<table>
<thead>
<tr>
<th>DR</th>
<th>No DR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smoothness($F_{sm}$)</td>
<td></td>
</tr>
<tr>
<td>Responsiveness($F_{r}$)</td>
<td></td>
</tr>
<tr>
<td>Stability($F_{st}$)</td>
<td></td>
</tr>
</tbody>
</table>

- Smoothness
- Responsiveness
- Stability
Experiments Results (RQ3)

✓ For responsiveness, the search with SM was 8 times faster

✓ For smoothness, the search with SM was much more effective
Our approach is able to identify critical violations of the controller requirements that had neither been found by our earlier work nor by manual testing.

<table>
<thead>
<tr>
<th></th>
<th>MiL-Testing different configurations</th>
<th>MiL-Testing fixed configurations</th>
<th>Manual MiL-Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stability</td>
<td>2.2% deviation</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Smoothness</td>
<td>24% over/undershoot</td>
<td>20% over/undershoot</td>
<td>5% over/undershoot</td>
</tr>
<tr>
<td>Responsiveness</td>
<td>170 ms response time</td>
<td>80 ms response time</td>
<td>50 ms response time</td>
</tr>
</tbody>
</table>
A Taxonomy of Automotive Functions

Computation

- Transforming
  - unit convertors

- Calculating
  - calculating positions, duty cycles, etc

Controlling

- State-Based
  - State machine controllers
- Continuous
  - Closed-loop controllers (PID)

Different testing strategies are required for different types of functions
Differences with Close-Loop Controllers

- Mixed discrete-continuous behavior: Simulink stateflows
- Much quicker simulation time
- No feedback loop -> no automated oracle
- The main testing cost is the manual analysis of output signals
- Goal: Minimize test suites
- Challenge: Test selection
- Entirely different approach to testing
Selection Strategies

• Adaptive Random Selection
• White-box Structural Coverage
  • State Coverage
  • Transition Coverage
• Output Diversity
• Failure-Based Selection Criteria (search)
  • Domain specific failure patterns
  • Output Stability
  • Output Continuity
Stability
Continuity
Minimizing CPU Shortage Risks During Integration

References:

Automotive: Distributed Development
Software Integration
Stakeholders

Car Makers
- Develop software optimized for their specific hardware
- Provide part suppliers with runnables (exe)

Part Suppliers
- Integrate car makers software with their own platform
- Deploy final software on ECUs and send them to car makers
Different Objectives

**Car Makers**

- Objective: Effective execution and synchronization of runnables
- Some runnables should execute simultaneously or in a certain order

**Part Suppliers**

- Objective: Effective usage of CPU time
- Max CPU time used by all the runnables should remain as low as possible over time
An overview of an integration process in the automotive domain

Original Equipment Manufacturer

AUTOSAR Models

AUTOSAR Models

sw runnables

Glue

sw runnables

DELPHI

Automotive Systems
CPU time shortage

- **Static cyclic scheduling:** predictable, analyzable
- **Challenge**
  - Many OS tasks and their many runnables run within a limited available CPU time
    - The execution time of the runnables may exceed their time slot
- **Our goal**
  - Reducing the maximum CPU time used per time slot to be able to
    - Minimize the hardware cost
    - Reduce the probability of overloading the CPU in practice
    - Enable addition of new functions incrementally

![CPU time usage simulation](image)
Using runnable offsets (delay times)

Offsets have to be chosen such that
the maximum CPU usage per time slot is minimized, and further,
the runnables respect their period
the runnables respect their time slot
the runnables satisfy their synchronization constraints
Without optimization

CPU time usage exceeds the size of the slot (5ms)
CPU time usage always remains less than 2.13ms, so more than half of each slot is guaranteed to be free.
Single-objective Search algorithms

Hill Climbing and Tabu Search and their variations

Solution Representation

a vector of offset values: o0=0, o1=5, o2=5, o3=0

Tweak operator

\[ o0=0, o1=5, o2=5, o3=0 \rightarrow o0=0, o1=5, o2=10, o3=0 \]

Synchronization Constraints

offset values are modified to satisfy constraints

Fitness Function

max CPU time usage per time slot
Summary of Problem and Solution

Optimization
while satisfying synchronization/temporal constraints

Explicit Time Model
for real-time embedded systems

Search
meta-heuristic single objective search algorithms

10^27
an industrial case study with a large search space
Search algorithms

- The objective function is the max CPU usage of a 2s-simulation of runnables
- The search modifies one offset at a time, and updates other offsets only if timing constraints are violated
- Single-state search algorithms for discrete spaces (HC, Tabu)

Case Study: an automotive software system with 430 runnables, search space = $10^{27}$

Running the system without offsets

Optimized offset assignment

5.34 ms

2.13 ms
Comparing different search algorithms

Best CPU usage

Time to find Best CPU usage
Comparing our best search algorithm with random search

(a) Lowest max CPU usage values computed by HC within 70 ms over 100 different runs.

(b) Lowest max CPU usage values computed by Random within 70 ms over 100 different runs.

(c) Comparing average behavior of Random and HC in computing lowest max CPU usage values within 70 s and over 100 different runs.

**HC**

**Random**

**Average**
Trade-off between Objectives

Car Makers

\[ r_0 \quad r_1 \quad r_2 \quad r_3 \]

Execute \( r_0 \) to \( r_3 \) close to one another.

Part Suppliers

Minimize CPU time usage

1 slot

0ms 5ms 10ms 15ms 20ms 25ms 30ms 4ms

2 slots

0ms 5ms 10ms 15ms 20ms 25ms 30ms 3ms

3 slots

0ms 5ms 10ms 15ms 20ms 25ms 30ms 2ms
Trade-off curve

- # of slots
- CPU time usage (ms)

Boundary Trade Offs

Interesting Solutions

- Point 1: (21, 1.45)
- Point 3: (14, 1.56)
- Point 2: (12, 2.04)
Multi-objective search

- Multi-objective genetic algorithms (NSGA II)
- Supporting decision making and negotiation between stakeholders

Objectives:
- (1) Max CPU time
- (2) maximum time slots between “dependent” tasks
Trade-Off Analysis Tool

Input.csv:
- runnables
- Periods
- CETs
- Groups
- # of slots per groups

Search

A list of solutions:
- objective 1 (CPU usage)
- objective 2 (# of slots)
- vector of group slots
- vector of offsets

Visualization/Query Analysis

- Visualize solutions
- Retrieve/visualize simulations
- Visualize Pareto Fronts
- Apply queries to the solutions
Conclusions

- Search algorithms to compute offset values that reduce the max CPU time needed
- Generate reasonably good results for a large automotive system and in a small amount of time
- Used multi-objective search tool for establishing trade-off between relaxing synchronization constraints and maximum CPU time usage
Schedulability Analysis and Stress Testing

References:

• S. Di Alesio et al., “Stress Testing of Task Deadlines: A Constraint Programming Approach”, IEEE ISSRE 2013, San Jose, USA
Real-time, concurrent systems (RTCS)

- Real-time, concurrent systems (RTCS) have concurrent interdependent tasks which have to finish before their deadlines
- Some task properties depend on the environment, some are design choices
- Tasks can trigger other tasks, and can share computational resources with other tasks
- How can we determine whether tasks meet their deadlines?
Problem

- **Schedulability analysis** encompasses techniques that try to predict whether all (critical) tasks are schedulable, i.e., meet their deadlines.
- **Stress testing** runs carefully selected test cases that have a high probability of leading to deadline misses.
- Stress testing is *complementary* to schedulability analysis.
- Testing is typically expensive, e.g., hardware in the loop.
- Finding stress test cases is difficult.
Finding Stress Test Cases is Difficult

\[ j_0, j_1, j_2 \text{ arrive at } at_0, at_1, at_2 \text{ and must finish before } dl_0, dl_1, dl_2 \]

\[ j_0, j_1, j_2 \]
\[ \text{at}_0 \]
\[ \text{dl}_0 \]
\[ \text{dl}_1 \]
\[ j_0, j_1, j_2 \]
\[ \text{at}_0 \]
\[ \text{dl}_0 \]
\[ \text{dl}_1 \]

\[ j_1 \text{ can miss its deadline } dl_1 \text{ depending on when } at_2 \text{ occurs!} \]
Challenges and Solutions

- Ranges for arrival times form a very large input space
- Task interdependencies and properties constrain what parts of the space are feasible
- We re-expressed the problem as a constraint optimisation problem
- Constraint programming (e.g., IBM CPLEX)
System monitors gas leaks and fire in oil extraction platforms

Drivers
(Software-Hardware Interface)

Control Modules

Real-Time Operating System

Multicore Architecture

Alarm Devices
(Hardware)
Constraint Optimization

**Constraint Optimization Problem**

- **Static Properties of Tasks**
  - (Constants)

- **Dynamic Properties of Tasks**
  - (Variables)

- **OS Scheduler Behaviour**
  - (Constraints)

- **Performance Requirement**
  - (Objective Function)
Process and Technologies

UML Modeling (e.g., MARTE)

Constraint Optimization

System Design

System Platform

Design Model (Time and Concurrency Information)

Deadline Misses Analysis

Optimization Problem
(Find arrival times that maximize the chance of deadline misses)

Constraint Programming (CP)

Stress Test Cases

Solutions
(Task arrival times likely to lead to deadline misses)

INPUT

OUTPUT
Challenges and Solutions

• Scalability problem: Constraint programming (e.g., IBM CPLEX) cannot handle such large input spaces (CPU, memory)

• Solution: Combine metaheuristic search and constraint programming
  – metaheuristic search identifies high risk regions in the input space
  – constraint programming finds provably worst-case schedules within these (limited) regions
  – Achieve (nearly) GA efficiency and CP effectiveness
Combining CP and GA

Fig. 3: Overview of GA+CP: the solutions $x_1$, $y_1$ and $z_1$ in the initial population of GA evolve into $x_6$, $y_6$, and $z_6$, then CP searches in their neighborhood for the optimal solutions $x^\star$, $y^\star$, and $z^\star$.

Let $J^\star(x)$ be the union of the impacting sets of tasks in $J(x)$:

$$I^\star(x) \defeq \left[ j^\star \in J^\star(x) \right]$$

By definition, $I^\star(x)$ contains all the tasks that can have an impact over a task that misses a deadline or is closest to a deadline miss.

Neighborhood $\varepsilon$ of an arrival time and neighborhood size $D$. Let $\varepsilon(x_{j,k})$ be the interval centered in the arrival time $x_{j,k}$ computed by GA, and let $D$ be its radius:

$$\varepsilon(x_{j,k}) = [x_{j,k}-D, x_{j,k}+D]$$

$\varepsilon$ defines the part of the search space around $x_{j,k}$ where to find arrival times that are likely to break task deadlines. $D$ is a parameter of the search.

Constraint Model $M$ implementing a Complete Search Strategy. Let $M$ be the constraint model defined in our previous work [Di Alesio et al. 2014] for the purpose of identifying arrival times for tasks that are likely to lead to deadline misses scenarios. $M$ models the static and dynamic properties of the software system respectively as constants and variables, and the scheduler of the operating system as a set of constraints among such variables. Note that $M$ implements a complete search strategy over the space of arrival times. This means that $M$ searches for arrival times of all aperiodic tasks within the whole interval $T$.

Our combined GA+CP strategy consists in the following two steps:
Process and Technologies

UML Modeling (e.g., MARTE)

Constraint Optimization

System Design
System Platform

Design Model (Time and Concurrency Information)

Deadline Misses Analysis

Optimization Problem
(Find arrival times that maximize the chance of deadline misses)

Genetic Algorithms (GA)
Constraint Programming (CP)

Solutions
(Task arrival times likely to lead to deadline misses)

Stress Test Cases

INPUT

OUTPUT
Environment-Based Testing of Soft Real-Time Systems

References:

Objectives

- Model-based system testing
  - Independent test team
  - Black-box
  - Environment models
Environment: Domain Model

- **Context**: RVM
  - notRoutingFlag : Boolean
  - "signal" user_inserts_item()
  - "signal" SUT_item_arrived()
  - "signal" ITEM_LOST()

- **Context**: User
  - count : integer
  - "NonDeterministic" insertionTime : Integer
  - "signal" rvm_sends_item()

- **Context**: Sorter
  - "NonDeterministic" moveArmTimeLC : Integer
  - "NonDeterministic" moveArmTimeCR : Integer
  - destination : String
    - "signal" POSITION_RIGHT()
    - "signal" POSITION_CENTRE()
    - "signal" POSITION_LEFT()
    - "signal" item_at_destination()

- **NonDeterministic**
  - Sorter::moveArmTimeLC {lowerBound = 280, upperBound = 320, scope = state}
  - Sorter::moveArmTimeCR {lowerBound = 280, upperBound = 320, scope = state}
Environment: Behavioral Model
Test Case Generation

- **Test objectives**: Reach “error” states (critical environment states)
- **Test Case**: (1) Environment and (2) Simulation Configuration
  - (1) Number of instances for each component in domain model, e.g., number of items on conveying belt
  - (2) Setting non-deterministic properties of the environment, e.g., speed of sorter’s left and right arms
- **Oracle**: Reaching an “error” state
- **Metaheuristics**: search for test cases getting to error state
- **Fitness function**
  - Distance from error state
  - Distance from satisfying guard conditions
  - Time distance
  - Time in “risky” states
Stress Testing focused on Concurrency Faults

Reference:

Stress Testing of Distributed Systems

Reference:

General Pattern: Using Metaheuristics

- Search to optimize objective function
- Metaheuristics, constraint programming
- Scalability: A small part of the search space is traversed
- Model: Guidance to worst case, high risk scenarios across space
- Reasonable modeling effort based on standards or extension
- Heuristics: Extensive empirical studies are required
Scalability
Project examples

- Scalability is the most common verification challenge in practice

- Testing closed-loop controllers
  - Large input and configuration space
  - Smart heuristics to avoid simulations (machine learning)

- Schedulability analysis and stress testing
  - Large space of possible arrival times
  - Constraint programming cannot scale by itself
  - CP was carefully combined with genetic algorithms
Scalability: Lessons Learned

- Scalability must be part of the problem definition and solution from the start, not a refinement or an after-thought.
- Meta-heuristic search, by necessity, has been an essential part of the solutions, along with, in some cases, machine learning, statistics, etc.
- Scalability often leads to solutions that offer “best answers” within time constraints, but no guarantees.
- Scalability analysis should be a component of every research project – otherwise it is unlikely to be adopted in practice.
- How many papers research papers do include even a minimal form of scalability analysis?

Applicability
Project examples

- Applicability requires to account for the domain and context
  - Testing controllers
    - Relies on Simulink only
    - No additional modeling or complex translation
    - Within domains, differences have huge implications in terms of applicability (open versus closed loop controllers)
  - Minimizing risks of CPU shortage
    - Trade-off between between effective synchronisation and CPU usage
    - Trade-off achieved through multiple objective GA search and appropriate decision tool
  - Schedulability analysis and stress testing
    - Near deadline misses must be identified
Applicability: Lessons Learned

• In software engineering, and verification in particular, just understanding the real problems in real contexts is difficult
• What are the inputs required by the proposed technique?
• How does it fit in development practices?
• Is the output what engineers require to make decisions?
• There is no unique solution to a problem as they tend to be context dependent, but a context is rarely unique and often representative of a domain
Discussion
Discussions

• **Metaheuristic search**
  – Tends to be versatile, easy to tailor to a new problem
  – Entails acceptable modeling requirements
  – Can provide “best” answers at any time
  – Scalable

**But**

– Not a proof, no certainty
– Though in practice (complex) models are not fully correct, there is no certainty anyway
– Effectiveness of search guidance is key and must be experimented and evaluated
– Models are key to provide adequate guidance
Discussion II

- **Constraint solvers (e.g., Comet, ILOG CPLEX, SICStus)**
  - Is there an efficient constraint model for the problem at hand?
  - Can effective heuristics be found to order the search?
  - Better if there is a match to a known standard problem, e.g., job shop scheduling
  - Tend to be strongly affected by small changes in the problem, e.g., allowing task pre-emption
  - Often not scalable, e.g., memory

- **Model checking**
  - Detailed operational models (e.g., state models), involving temporal properties (e.g., CTL)
  - Enough details to analyze statically or execute symbolically
  - These modeling requirements are usually not realistic in actual system development. State explosion problem.
  - Originally designed for checking temporal properties through reachability analysis, as opposed to explicit timing properties
  - Often not scalable
Talk Summary

• Focus: Meta-heuristic Search to enable scalable verification and testing.
• Scalability is the main challenge in practice.
• Drew lessons learned from example projects in collaboration with industry, on real systems and in real verification contexts.
• Results show that meta-heuristic search contributes to mitigate the scalability problem.
• It has shown to lead to applicable solutions in practice.
• Solutions are very context dependent.
• It is usually combined with a variety of other complementary techniques: system modeling, constraint solving, machine learning, statistics.
Scalable Software Testing and Verification Through Heuristic Search and Optimization

Lionel Briand

Interdisciplinary Centre for ICT Security, Reliability, and Trust (SnT)
University of Luxembourg, Luxembourg

GDR-GPL, Bordeaux, June 2015

SVV lab: svv.lu
SnT: www.securityandtrust.lu