Gaining Confidence in the Correctness of Robotic and Autonomous Systems

Kerstin Eder

Design Automation and Verification
Trustworthy Systems, University of Bristol

Verification and Validation for Safety in Robots, Bristol Robotics Laboratory
Would you swallow a robot?
Figure 11 — Hierarchy of Verification and Validation activities

Figure 11 defines the complete nature of verification and validation (V&V) activities. V&V can be done on system, hardware, and software products. These activities and planning are defined and refined in IEEE 1012 and ISO/IEC 12207. Much of V&V is accomplished by testing. The ISO/IEC 29119 standard addresses the Dynamic and Static software testing (directly or via reference), thus covering parts of this verification and validation model. ISO/IEC 29119 is not intended to address all the elements of the V&V model, but it is important for a tester to understand where they fit within this model.
The Safety Challenge

- Autonomous Systems
- Engineering Challenge
  - Advances in control engineering and ML
  - Focus on “making things work”
The Safety Challenge

- Autonomous Systems
- Engineering Challenge
  - Advances in control science
  - Focus on “making things work”
- Fundamental concern:
  - Can such systems be trusted?
Designing Trustworthy Systems

- Create flawless systems.

AND

- Design these systems in such a way that the flawlessness can be demonstrated.

"Waterfall" by M.C. Escher.
“Robots are products. They should be **designed using processes which assure their safety** and security.”

http://www.epsrc.ac.uk/ourportfolio/themes/engineering/activities/Pages/principlesofrobotics.aspx
Verification and Validation for Safety in Robots

To develop techniques and methodologies that can be used to design autonomous intelligent systems that are verifiably trustworthy.
Correctness from specification to implementation

User Requirements
High-level Specification

Translate

Optimizer
Design and Analysis (Simulink)

Implement

Controller (SW/HW)
e.g. C, C++, RTL (VHDL/Verilog)
What can be done at the code level?

P. Trojanek and K. Eder.  
*Verification and testing of mobile robot navigation algorithms: A case study in SPARK.*  
http://dx.doi.org/10.1109/IROS.2014.6942753
What can go wrong in robot navigation software?

Generic bugs:
- Array and vector out-of-bounds accesses
- Null pointer dereferencing
- Accesses to uninitialized data

Domain-specific bugs:
- Integer and floating-point arithmetic errors
- Mathematic functions domain errors
- Dynamic memory allocation errors
- Concurrency bugs and blocking inter-thread communication (non real-time)
Navigation in SPARK

- Three open-source implementations of navigation algorithms originally in C/C++ (2.7 kSLOC)
  - Vector Field Histogram
  - Nearness Diagram
  - Smooth Nearness-Diagram

<table>
<thead>
<tr>
<th></th>
<th>Driver C++</th>
<th>Algorithm C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFH+</td>
<td>807</td>
<td>782</td>
</tr>
<tr>
<td>ND</td>
<td>828</td>
<td>1037</td>
</tr>
<tr>
<td>SND</td>
<td>403</td>
<td>941</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2038</strong></td>
<td><strong>2760</strong></td>
</tr>
</tbody>
</table>
Verification Approach

State of the art verification approaches:
- Model checking: infeasible
- Static analysis of C++: not possible
- Static analysis of C: requires verbose and difficult to maintain annotations

A Design-for-Verification approach:
- **SPARK**, a verifiable subset of Ada
  - software reliability a primary goal
  - SPARK specification and tools free for academic use
- Required code modifications:
  - Pre- and post-conditions, loop (in)variants
  - Numeric subtypes (e.g. Positive)
  - Formal data containers
Navigation in SPARK

- Three open-source implementations of navigation algorithms translated from C/C++ (2.7 kSLOC) to SPARK (3.5 kSLOC)
  - Vector Field Histogram
  - Nearness Diagram
  - Smooth Nearness-Diagram

<table>
<thead>
<tr>
<th></th>
<th>Driver</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C++</td>
<td>C/C++</td>
</tr>
<tr>
<td>VFH+</td>
<td>807</td>
<td>782</td>
</tr>
<tr>
<td>ND</td>
<td>828</td>
<td>1037</td>
</tr>
<tr>
<td>SND</td>
<td>403</td>
<td>941</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2038</strong></td>
<td><strong>2760</strong></td>
</tr>
</tbody>
</table>

- Explicit annotations are less than 5% of the code
- SPARK code is on average 30% longer than C/C++
Verification Conditions

<table>
<thead>
<tr>
<th></th>
<th>Pre-conditions</th>
<th>Post-conditions</th>
<th>Loop invariants*</th>
<th>Loop variants</th>
<th>Assertions</th>
<th>Divisions</th>
<th>Integer overflows</th>
<th>Floating-point overflows</th>
<th>Subtype ranges</th>
<th>Array indices</th>
<th>Record discriminants</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VFH+</strong></td>
<td>46 (3)</td>
<td>5</td>
<td>18 (9)</td>
<td>0</td>
<td>23</td>
<td>36</td>
<td>36</td>
<td>120</td>
<td>100</td>
<td>102</td>
<td>262</td>
<td>748</td>
</tr>
<tr>
<td><strong>ND</strong></td>
<td>83 (18)</td>
<td>10</td>
<td>8 (4)</td>
<td>2</td>
<td>3</td>
<td>54</td>
<td>23</td>
<td>254</td>
<td>53</td>
<td>50</td>
<td>0</td>
<td>540</td>
</tr>
<tr>
<td><strong>SND</strong></td>
<td>104 (9)</td>
<td>9</td>
<td>14 (7)</td>
<td>2</td>
<td>30</td>
<td>29</td>
<td>1</td>
<td>140</td>
<td>22</td>
<td>0</td>
<td>24</td>
<td>375</td>
</tr>
</tbody>
</table>

* Separate verification conditions are generated for each call to subprogram with precondition, and similarly for initialization and preservation of each loop invariant; the numbers of explicit annotations are given in brackets.
Formal Verification Outcome

<table>
<thead>
<tr>
<th></th>
<th>Alt-Ergo 0.96</th>
<th>Z3 4.3.1</th>
<th>Alt-Ergo &amp; Z3 combined</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFH+</td>
<td>633</td>
<td>699</td>
<td>701</td>
<td>748</td>
</tr>
<tr>
<td></td>
<td>11 min</td>
<td>37 min</td>
<td>48 min</td>
<td></td>
</tr>
<tr>
<td>ND</td>
<td>462</td>
<td>482</td>
<td>483</td>
<td>540</td>
</tr>
<tr>
<td></td>
<td>17 min</td>
<td>21 min</td>
<td>41 min</td>
<td></td>
</tr>
<tr>
<td>SND</td>
<td>350</td>
<td>366</td>
<td>366</td>
<td>375</td>
</tr>
<tr>
<td></td>
<td>29 min</td>
<td>6 min</td>
<td>36 min</td>
<td></td>
</tr>
</tbody>
</table>

Number of discharged verification conditions and the running time of static analysis based on two SMT solvers, Alt-Ergo and Z3.
Results

- Several bugs discovered by run-time checks injected by the Ada compiler
  - Fixed code proved to be run-time safe
    - except floating-point over- and underflows
    - These require the use of complementary techniques, e.g. abstract interpretation.

- Up to 97% of the verification conditions discharged automatically by SMT solvers in less than 10 minutes

- Performance of the SPARK and C/C++ code similar
Moral

If you want to make runtime errors an issue of the past, then you must select your tools (programming language and development environment) wisely!

http://github.com/riveras/spark-navigation

P. Trojanek and K. Eder.  
*Verification and testing of mobile robot navigation algorithms: A case study in SPARK.*  
http://dx.doi.org/10.1109/IROS.2014.6942753
Correctness from Specification to Implementation

User Requirements
High-level Specification

Optimizer
Design and Analysis (Simulink)

Controller (SW/HW)
e.g. C, C++, RTL (VHDL/Verilog)
What can be done at the design level?

*Formal Verification of Control Systems’ Properties with Theorem Proving.*  
International Conference on Control (CONTROL), pp. 244 - 249. IEEE, Jul 2014.  
[http://dx.doi.org/10.1109/CONTROL.2014.6915147](http://dx.doi.org/10.1109/CONTROL.2014.6915147)

*Verification of Control Systems Implemented in Simulink with Assertion Checks and Theorem Proving: A Case Study.*  
Simulink Diagrams in Control Systems

Simulating the control systems
- Principles of control systems theory (e.g., stability)
- Serve as requirements/specification
- For (automatic) code generation

\[ x(k+1) = Ax(k) + Bu(k) \]
\[ u(k) = -Kx(k) \]
Verifying Stability

Matrix $\mathbf{P} > 0$ (Lyapunov function)

Matrix $\mathbf{P} - (\mathbf{A} - \mathbf{BK})^T \mathbf{P}(\mathbf{A} - \mathbf{BK}) > 0$ (Lyapunov function's difference)

Equivalence

$V(k) - V(k-1) = x(k-1)^T [(\mathbf{A} - \mathbf{BK})^T \mathbf{P}(\mathbf{A} - \mathbf{BK}) - \mathbf{P}] x(k-1)$ (Lyapunov's equation application)

Capture control systems requirements

Add as assertions

Retain in code implementation
Assertion-Based Verification
Combining Verification Techniques

- **Stability**
  - Matrix $\mathbf{P} > 0$ (Lyapunov function)
  - Matrix $\mathbf{P} - (\mathbf{A} - \mathbf{B}\mathbf{K})^T \mathbf{P}(\mathbf{A} - \mathbf{B}\mathbf{K}) > 0$ (Lyapunov function's difference)
  - Equivalence $V(k) - V(k-1) = x(k-1)^T [(\mathbf{A} - \mathbf{B}\mathbf{K})^T \mathbf{P}(\mathbf{A} - \mathbf{B}\mathbf{K}) - \mathbf{P}] x(k-1)$ (Lyapunov's equation application)

- Test in simulation
- First order logic theory of the Simulink diagram
  - Axiom: $\mathbf{B}u = \mathbf{B} \ast u$
  - Goal: $\mathbf{vdiff} == \mathbf{vdiff}_{\text{an}}$
- Automatic theorem proving
No single technique is adequate to cover a whole design in practice. Combine techniques and learn from areas where verification is more mature.
http://github.com/riveras/simulink

**Formal Verification of Control Systems’ Properties with Theorem Proving.** International Conference on Control (CONTROL), pp. 244 - 249. IEEE, Jul 2014.  
http://dx.doi.org/10.1109/CONTROL.2014.6915147

**Verification of Control Systems Implemented in Simulink with Assertion Checks and Theorem Proving: A Case Study.**  
http://arxiv.org/abs/1505.05699
What can be done to advance simulation-based testing of RAS?


Robot to human hand-over task

When should the robot let go, i.e. when is it safe for the robot to let go?
We are investigating…

- Testing in simulation
- Coverage-Driven Verification (CDV), a technique well established in microelectronics design verification

… to verify code that controls robots in HRI.
CDV to automate simulation-based testing


Dejanira Araiza-Illan, David Western, Anthony Pipe and Kerstin Eder.  
Robotic code

Simulation-based testing

Dejanira Araiza-Illan, David Western, Anthony Pipe and Kerstin Eder.
Test Generator

- Effective tests:
  - legal tests
  - meaningful events
  - interesting events
  - while exploring the system
    - typical vs extreme values

- Efficient tests:
  - minimal set of tests (regression)

- Strategies:
  - Pseudorandom (repeatability)
Test Generator

- **Effective tests:**
  - legal tests
  - meaningful events
  - interesting events
  - while exploring the system
    - typical vs extreme values

- **Efficient tests:**
  - minimal set of tests (regression)

- **Strategies:**
  - Pseudorandom (repeatability)
  - Constrained pseudorandom
Model-based test generation

Formal model → Traces from model checking → Test template → Test components:
- High-level action sequence
- Parameter instantiation

System + environment → Environment to drive system
Model-based test generation

Formal model → Traces from model checking → Test template → Test components:
- High-level action sequence
- Parameter instantiation

System + environment

Environment to drive system
Simulation-based testing

Dejanira Araiza-Illan, David Western, Anthony Pipe and Kerstin Eder. 
Encode requirements as assertions:
- if [precondition], check [postcondition]

“If the robot decides the human is not ready, then the robot never releases an object”.

- Implemented as automata for monitoring

Continuous monitoring at runtime, self-checking
- High-level requirements
- Lower-level requirements depending on the simulation's detail (e.g., path planning, collision avoidance).

assert {! (robot_3D_position == human_3D_position)}
Simulation-based testing

Dejanira Araiza-Illan, David Western, Anthony Pipe and Kerstin Eder.  
Coverage

- Code coverage
- **Structural coverage**, e.g. of the FSM(s)
- **Functional coverage**
  - Requirements coverage
    - Functional and safety (ISO 13482:2014, ISO 10218-1)
Robot to human object handover scenario
Robot to human object handover scenario
Requirements inspired by ISO 13482 and ISO 10218

1. If the gaze, pressure and location are sensed as correct, then the object shall be released.
2. If the gaze, pressure or location are sensed as incorrect, then the object shall not be released.
3. The robot shall make a decision before a threshold of time.
4. The robot shall always either time out, decide to release the object, or decide not to release the object.
5. The robot shall not close the gripper when the human is too close.
6. The robot shall start in restricted speed and force.
7. The robot shall not collide with itself at high speeds.
8. The robot shall operate within allowable maximum values to avoid dangerous unintentional collisions with humans and other safety-related objects.
Requirements inspired by ISO 13482 and ISO 10218

1. If the gaze, pressure and location are sensed as correct, then the object shall be released.
2. If the gaze, pressure or location are sensed as incorrect, then the object shall not be released.
3. The robot shall make a decision before a threshold of time.
4. The robot shall always either time out, decide to release the object, or decide not to release the object.
5. The robot shall not close the gripper when the human is too close.
6. The robot shall start in restricted speed and force.
7. The robot shall not collide with itself at high speeds.
8. The robot shall operate within allowable maximum values to avoid dangerous unintentional collisions with humans and other safety-related objects.
Requirements inspired by ISO 13482 and ISO 10218

Considering a speed threshold of 250 mm/s (from ISO 10218-1), last requirement implemented as:

3a. The robot hand speed is always less than 250 mm/s.

3b. If the robot is within 10 cm of the human, the robot’s hand speed is less than 250 mm/s.

3c. If the robot collides with anything, the robot’s hand speed is less than 250 mm/s.

3d. If the robot collides with the human, the robot’s hand speed is less than 250 mm/s.
Coverage

- Code coverage
- **Structural coverage**, e.g. of the FSM(s)
- **Functional coverage**
  - Requirements coverage
    - Functional and safety (ISO 13482:2014, ISO 10218-1)
  - Situation coverage (cross-product coverage) based on gaze, pressure and hand location sensor data
Coverage

- **Code coverage**
- **Structural coverage**, e.g. of the FSM(s)
- **Functional coverage**
  - Requirements coverage
    - Functional and safety (ISO 13482:2014, ISO 10218-1)
  - Situation coverage (cross-product coverage) based on gaze, pressure and hand location sensor data
  - SOTIF
    (ISO/PAS 21448:2019)

Road vehicles, Safety of the intended functionality
CDV for Human-Robot Interaction

Dejanira Araiza-Illan, David Western, Anthony Pipe and Kerstin Eder.

Coverage-Directed Verification

- systematic, goal directed verification method
  - high level of automation
  - capable of exploring systems of realistic detail under a broad range of environment conditions
- focus on test generation and coverage
  - constraining test generation requires significant engineering skill and SUT knowledge
  - model-based test generation allows targeting requirements and cross-product coverage more effectively than pseudorandom test generation
Dejanira Araiza-Illan, David Western, Anthony Pipe and Kerstin Eder.  
*Coverage-Driven Verification — An Approach to Verify Code for Robots that Directly Interact with Humans.* In Hardware and Software: Verification and Testing, pp. 69-84. Lecture Notes in Computer Science 9434. Springer, November 2015. (DOI: [10.1007/978-3-319-26287-1_5](https://doi.org/10.1007/978-3-319-26287-1_5))

Dejanira Araiza-Illan, David Western, Anthony Pipe and Kerstin Eder.  
CDV provides automation

What about agency?
Robotic assistants need to be both powerful and *smart*.

– AI and learning are increasingly used in robotics

We need *intelligent* testing.

– No matter how clever your robot, the testing environment needs to reflect the *agency* your robot will meet in its target environment.
INITIAL BELIEFS
preparing_for_flight
initialising_systems
-hardware_system_passed_test
-has_read_flight_environment_model
-has_read_new_flight_path
-pilot_comms_work
-all Beacon_comms_work
-created_flight_path_execution_plan
-plan_is_unsafe_for_energy_level_available(Flight)
-announced_text_object
-ready_for_mission
-on_ground_before_flight
-ground_testing
-ground_based
-responded_to_take_off_permission
-permission_given_for_take_off
-flying
-taking_off_testing
-Flight_system_weakness_to_report
-ready_for_start_mission
-on_mission
-people_paused
-vehicle_paused
-flying_paused
-avoiding_behaviour
-power_return
-emergency_Landing
-in_manual_control
-landed

Environment Events and States
people_appearing
vehicles_appearing
flying_object_appearing
weather_too_bad
visibility_too_bad
onboard_faults
command_received
manual_control_request
Belief-Desire-Intention Agents

Desires: goals to fulfil

Beliefs: knowledge about the world

Intentions: chosen plans, according to current beliefs and goals

New goals

New beliefs

Guards for plans

From executing plans
Intelligent testing is harnessing the power of BDI agent models to introduce agency into test environments.
Research Questions

- Are Belief-Desire-Intention agents suitable to model HRI?
- How can we exploit BDI agent models for test generation?
- Can machine learning be used to automate test generation in this setting?
- How do BDI agent models compare to automata-based techniques for model-based test generation?
Interacting Agents

- BDI can model agency in HRI
  - Interactions between agents create realistic action sequences that serve as test patterns
Interacting Agents

- BDI can model agency in HRI
  - Interactions between agents create realistic action sequences that serve as test patterns
Verification Agents

- Meta agents can influence beliefs
- This allows biasing/directing the interactions
Which beliefs are effective?

(Meta Agent) Verification Agent

Agent for Simulated Human

Agents for Simulated Sensors

Robot’s Code Agent

Manual belief selection

belief subsets
Which beliefs are effective?

Manual belief selection
Random belief selection

(Meta Agent) Verification Agent

Agent for Simulated Human

Agent for Simulated Sensors

Robot’s Code Agent

belief subsets

beliefs

beliefs

beliefs

beliefs

beliefs

beliefs

beliefs
Which beliefs are effective?

Optimal belief sets determined through RL

plan coverage

belief subsets

(Meta Agent) Verification Agent

Agent for Simulated Human

Agents for Simulated Sensors

Robot's Code Agent

Optimal belief sets determined through RL

belief subsets
Results

How effective are BDI agents for test generation? How do they compare to model checking timed automata?


D. Araiza-Illan, A.G. Pipe, K. Eder
The cost of learning belief sets

The cost of learning a good belief set needs to be considered when assessing the different BDI-based test generation approaches.

Convergence in <300 iterations, < 3 hours
Code Coverage Results
Effectiveness:
- high-coverage tests are generated quickly
BDI-agents vs timed automata
BDI-agents vs timed automata

<table>
<thead>
<tr>
<th></th>
<th>Model checking timed automata</th>
<th>BDI agents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cooperative Manufacturing Assistant</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model’s lines of code</td>
<td>725</td>
<td>348</td>
</tr>
<tr>
<td>Number of states (transitions) or plans</td>
<td>53 (72)</td>
<td>79</td>
</tr>
<tr>
<td>Modelling time</td>
<td>(\approx 10.5) hrs</td>
<td>(\approx 6) hrs</td>
</tr>
<tr>
<td>Model exploration time (min/test)</td>
<td>0.001 s</td>
<td>5 s</td>
</tr>
<tr>
<td>Model exploration time (max/test)</td>
<td>33.36 s</td>
<td>5 s</td>
</tr>
<tr>
<td><strong>Home Care Assistant</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model’s lines of code</td>
<td>722</td>
<td>131</td>
</tr>
<tr>
<td>Number of states (transitions) or plans</td>
<td>42 (67)</td>
<td>35</td>
</tr>
<tr>
<td>Modelling time</td>
<td>(\approx 5.5) hrs</td>
<td>(\approx 3) hrs</td>
</tr>
<tr>
<td>Model exploration time (min/test)</td>
<td>0.001 s</td>
<td>1 s</td>
</tr>
<tr>
<td>Model exploration time (max/test)</td>
<td>2.775 s</td>
<td>1 s</td>
</tr>
</tbody>
</table>
Belief-Desire-Intention agents are suitable to model HRI.

Traces of interactions between BDI agent models provide test templates.

Machine learning (RL) can be used to automate the selection of belief sets so that test generation can be biased towards maximizing coverage.

Compared to traditional model-based test generation (model checking timed automata), BDI models are:

- more intuitive to write, they naturally express agency,
- smaller in terms of model size,
- more predictable to explore and
- equal if not better wrt coverage.
http://github.com/robosafe

D. Araiza Illan, D. Western, A. Pipe, K. Eder.

D. Araiza Illan, D. Western, A. Pipe, K. Eder.

DOI: 10.1145/3022099.3022101 (arXiv:1604.05508)

D. Araiza-Illan, A.G. Pipe, K. Eder
Challenges for RAS V&V

- Specification: vague and probabilistic
  J. Morse, D. Araiza-Illan, J. Lawry, A. Richards, K. Eder

- Automation, automation, automation

- Combination of techniques

- More AI for V&V, ... *we* need to be more clever
  - Intelligent agent-based test generation:
    - a step towards online testing of learning machines
    - testing games between verification agents and robots
Thank you

Kerstin.Eder@bristol.ac.uk

Special thanks to Dejanira Araiza Illan, Jeremy Morse, David Western, Greg Chance, Abanoub Ghobrial, Arthur Richards, Jonathan Lawry, Trevor Martin, Clare Dixon, Michael Fisher, Matt Webster, Kerstin Dautenhahn, Maha Salem, Piotr Trojanek, Yoav Hollander, Yaron Kashai, Mike Bartley, Séverin Lemaignan, Tony Pipe and Chris Melhuish for their collaboration, contributions, inspiration and the many productive discussions we have had.