Combined use of formal methods for reliability assurance of software for safety-critical systems

Igor Buzhinskii

Doctoral dissertation

Supervisors: Prof. Valeriy Vyatkin (Aalto University),
Prof. Anatoly Shalyto (ITMO University)

Opponent: Prof. Jean-Jacques Lesage (ENS Paris-Saclay)

28.06.2019, Espoo
Motivation: reliable control of safety-critical systems

Industrial **plant**

IoT devices, transport, power plants, ...

Might be **dangerous** to people and the environment

**Controller** (e.g. PLC)
Motivation: reliable control of safety-critical systems

Industrial plant

- IoT devices, transport, power plants, ...
- Might be **dangerous** to people and the environment
- Thus, must be **reliably** controlled to achieve **safety** while **doing useful work**

Controller (e.g. PLC)

Traditional approaches: testing and simulation
Motivation: reliable control of safety-critical systems

Industrial plant

IoT devices, transport, power plants, ...

Might be dangerous to people and the environment

Thus, must be reliably controlled to achieve safety while doing useful work

Traditional approaches: testing and simulation

Thorough approaches: formal verification and formal synthesis
• In Finland, a formal verification method called **model checking** is used to ensure the safety of nuclear power plants.
In Finland, a formal verification method called **model checking** is used to ensure the safety of nuclear power plants.

Specifically, VTT model-checks instrumentation and control (I&C) systems for Finnish power utilities.
Finite-state machines and other kinds of formal models

States = memory

Transitions & guard conditions

Outputs

¬last

last

¬INPUT / ¬OUTPUT

INPUT / ¬OUTPUT

¬INPUT / ¬OUTPUT

INPUT / OUTPUT

¬INPUT / OUTPUT
Finite-state machines and other kinds of formal models

States = memory

Transitions & guard conditions

Outputs

Textual formal languages

MODULE DELAY (INPUT)
VAR
  last: boolean;
ASSIGN
  init(last) := FALSE;
  next(last) := INPUT;
DEFINE
  OUTPUT := last;

Combined use of formal methods for reliability assurance of software for safety-critical systems 28.06.2019 4 / 25
Finite-state machines and other kinds of formal models

- States = memory
- Transitions & guard conditions
- Outputs

Textual formal languages

```
MODULE DELAY (INPUT)
VAR
  last: boolean;
ASSIGN
  init(last) := FALSE;
  next(last) := INPUT;
DEFINE
  OUTPUT := last;
```

Networks of formal models
Finite-state machines and other kinds of formal models

States = memory

Transitions & guard conditions

Outputs

Textual formal languages

MODULE DELAY (INPUT)

VAR last : boolean;

ASSIGN init(last) := FALSE;

next(last) := INPUT;

DEFINE OUTPUT := last;

Model checking: automatic and thorough behavior analysis

Networks of formal models

Combined use of formal methods for reliability assurance of software for safety-critical systems
Expectation

Verification tool  X

System

Specification:
“The system shall work correctly and safely”

Check! ☀️

OK

Not OK, because <explanation that you can easily understand>
Model checking in practice

Expectation

Verification tool: X

System: 

Specification: "The system shall work correctly and safely"

Check! → OK

Not OK, because <explanation that you can easily understand>

Reality
Model checking in practice

**Expectation**

<table>
<thead>
<tr>
<th>Verification tool</th>
<th>X</th>
</tr>
</thead>
</table>

- **System**
- **Specification:** “The system shall work correctly and safely”

**Check!**

**Reality**

- **How to construct models?**
  - Plant and controller

OK

**Not OK**, because

<explanation that you can easily understand>
Model checking in practice

<table>
<thead>
<tr>
<th>Expectation</th>
<th>Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification tool</td>
<td>How to construct models? Plant and controller</td>
</tr>
<tr>
<td>System</td>
<td>Computationally expensive! Verification time grows exponentially with the growth of the model</td>
</tr>
<tr>
<td>Specification: “The system shall work correctly and safely”</td>
<td></td>
</tr>
<tr>
<td>Check!</td>
<td></td>
</tr>
</tbody>
</table>

OK
Not OK, because <explanation that you can easily understand>
Model checking in practice

**Expectation**

- Verification tool: \( \times \)
- System
- Specification: "The system shall work correctly and safely"

**Check!**

- OK
- Not OK, because <explanation that you can easily understand>

**Reality**

- How to construct models?
- Computationally expensive!
- Verification time grows exponentially with the growth of the model
- Difficult to formalize specifications
- Difficult to interpret the results of verification
Model checking in practice

**Expectation**

<table>
<thead>
<tr>
<th>Verification tool</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Specification: “The system shall work correctly and safely”</td>
</tr>
<tr>
<td>Check!</td>
<td>![sun]</td>
</tr>
</tbody>
</table>

**Reality**

- **How to construct models?**
  - Plant and controller

- **Computationally expensive!**
  - Verification time grows exponentially with the growth of the model

- Difficult to formalize specifications
- Difficult to interpret the results of verification
- Many things are not automated → human factor (mistakes are possible)

Not OK, because <explanation that you can easily understand>
Goal: increase the **efficiency**, **effectiveness**, and the **level of automation** of reliability assurance of safety-critical control systems

<table>
<thead>
<tr>
<th>Contribution</th>
<th>Efficiency</th>
<th>Effectiveness</th>
<th>Automation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contribution 1</td>
<td>✔️ ✔️ ✔️</td>
<td>✔️ ✔️ ✔️</td>
<td>✔️ ✔️ ✔️</td>
</tr>
<tr>
<td>Contribution 2</td>
<td>✔️ ✔️ ✔️</td>
<td>✔️ ✔️ ✔️</td>
<td>✔️ ✔️ ✔️</td>
</tr>
<tr>
<td>Contribution 3</td>
<td>✔️ ✔️ ✔️</td>
<td>✔️ ✔️ ✔️</td>
<td>✔️ ✔️ ✔️</td>
</tr>
</tbody>
</table>
• Contribution 1: methods to generate plant and controller models from specification

• Contribution 2: method to verify formal models created manually based on a natural language specification

• Contribution 3: testing method for closed-loop systems based on model checking
Overview of proposed model generation methods

- Idea: generate the components of the closed-loop system from specification

Behavior examples (traces)

- level = 100
  - pressure = 200

- level = 110
  - pressure = 180

- level = 120
  - pressure = 170

...
Overview of proposed model generation methods

- Idea: generate the components of the closed-loop system from specification

Behavior examples (traces)

- Level = 100
  - Pressure = 200
- Level = 110
  - Pressure = 180
- Level = 120
  - Pressure = 170

LTL properties

- Always, if level < 200, then eventually pressure > 150
Overview of proposed model generation methods

- Idea: generate the components of the closed-loop system from specification

Behavior examples (traces)

<table>
<thead>
<tr>
<th>Level (mm)</th>
<th>Pressure (mmHg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>110</td>
<td>180</td>
</tr>
<tr>
<td>120</td>
<td>170</td>
</tr>
</tbody>
</table>

LTL properties

- \(\text{always, if} \ \text{level} < 200, \ \text{then eventually} \ \text{pressure} > 150\)

Plant or controller state machine

- \(50 < \text{level} \leq 100\)
- \(190 < \text{pressure} \leq 200\)

- \(100 < \text{level} \leq 150\)
- \(150 < \text{pressure} \leq 190\)

- Open valve
- Close valve
Example of a constructed controller model for an ATM

- Initially, the state machine satisfies only positive behavior examples

Card inserted; request PIN

Authorization request; send PIN to the server

https://www.flickr.com/photos/osipovva/37956739554/in/photostream/
"ATM" by valeriy osipov is licensed under CC BY 2.0
Initially, the state machine satisfies only \textbf{positive behavior examples}.

Iteratively, it is refined with \textbf{negative examples}.

---

Card inserted; request PIN

Authorization request; send PIN to the server
Example of a constructed controller model for an ATM

- Initially, the state machine satisfies only positive behavior examples
- Iteratively, it is refined with negative examples
- In the end, the state machine satisfies LTL properties
Example of a constructed controller model for an ATM

- Initially, the state machine satisfies only positive behavior examples
- Iteratively, it is refined with negative examples
- In the end, the state machine satisfies LTL properties
- All is done with Boolean formula solving
Scalable plant model construction methods

- Idea: generate large plant models quickly

Plant behavior examples (traces)

level = 100
pressure = 200

level = 110
pressure = 180

level = 120
pressure = 170

... ... ...

always, if level < 200, then eventually pressure > 150
Scalable plant model construction methods

- Idea: generate large plant models quickly

Plant behavior examples (traces)

```
level = 100
pressure = 200
```

```
level = 110
pressure = 180
```

```
level = 120
pressure = 170
```

... ... ...

always, if level < 200, then eventually pressure > 150
Scalable plant model construction methods

- Idea: generate large plant models quickly

Plant behavior examples (traces)

level = 100
pressure = 200

level = 110
pressure = 180

level = 120
pressure = 170

...always, if level < 200, then eventually pressure > 150

State machine

System of constraints

MODULE PLANT
VAR level: 0..200;
VAR pressure: 0..250;
TRANS level >= 100 ->
  next(level) > 100;
TRANS pressure < 200 ->
  level > 105;
Scalable plant model construction methods

- Idea: generate large plant models quickly

Plant behavior examples (traces)

- always, if level < 200, then eventually pressure > 150

State machine

System of constraints

- MODULE PLANT
  VAR level: 0..200;
  VAR pressure: 0..250;
  TRANS level >= 100 -> 
  next(level) > 100;
  TRANS pressure < 200 -> 
  level > 105;

Modular model
Fragment of a plant model for a nuclear power plant

Blue = low reactor power
White = medium reactor power
Red = high reactor power
Two methods of **plant and controller** model generation based on **both** behavior examples and LTL properties

- Generation of **minimum models**, which is especially important for controller models
- High computational complexity

Three **scalable** methods of **plant** model generation based on **only** behavior examples

- Time complexity is **linear** of input size
- Support of **modular** plant models
- Scalability and modularity are important due to the **large size** of real-world plants

**Contribution 1**

<table>
<thead>
<tr>
<th>Efficiency</th>
<th>Effectiveness</th>
<th>Automation</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Contributions 2/3

• Contribution 1: methods to generate plant and controller models from specification

• Contribution 2: method to verify formal models created manually based on a natural language specification

• Contribution 3: testing method for closed-loop systems based on model checking
Natural language (textual) specification

- **Pulse** module can be used as a pulse generator in a logical circuit. The module will generate pulses of a given length every time when the input signal state is changing from FALSE to TRUE...

- ... “Mode 2: if the input signal’s value changes to FALSE and back to TRUE during a pulse, the pulse time stays as defined.” ...

```
IN: bool
```

```
OUT: bool
```

```
pulse pulse
```

```
IN: bool
```

```
OUT: bool
```
Example of basic block: Pulse (Apros simulator)

Natural language (textual) specification

- "**Pulse** module can be used as a pulse generator in a logical circuit. The module will generate pulses of a given length every time when the input signal state is changing from FALSE to TRUE…"

- "…Mode 2: if the input signal’s value changes to FALSE and back to TRUE during a pulse, the pulse time stays as defined.” …

IN: bool

OUT: bool

---

```
MODULE PULSE(PULSE_INPUT_SIGN, PULSE_LENGTH, PULSE_MODE)
VAR
  timeLeft: -1..200;
  lastInput: boolean;
  lastOutput: boolean;
DEFINE
  PULSE_OUTPUT_SIGN := case
    PULSE_MODE = 0: !lastInput & PULSE_INPUT_SIGN;
    ...
    PULSE_MODE = 2: !lastOutput & !lastInput & PULSE_INPUT_SIGN
      | timeLeft > 0;
    ...
```

---

manually created model

Combined use of formal methods for reliability assurance of software for safety-critical systems
Idea of the proposed method: cycle of refinement

Manually created model

1. Manual formalization

Textual specification

LTL specification

Input artifacts

Manually created model

Manually created model

\[
\text{SW: BINARY SWJ} \\
\text{MINUS, ADDER, OL} \\
\text{PLUS: ADDER_3} \\
\text{MINUS: ADDER_3} \\
\text{OUT := SW, BSW} \\
\]
Idea of the proposed method: cycle of refinement

1. Manual formalization
   - Textual specification
     - LTL specification
2. Generation
   - Generated model
   - Input artifacts
     - Manually created model

Combined use of formal methods for reliability assurance of software for safety-critical systems
Idea of the proposed method: cycle of refinement

1. Manual formalization
   - Textual specification

2. Generation
   - LTL specification

3. Equivalence model-check
   - Generated model
   - Example of differing behaviors

Output: log of faults in the model and specification ambiguities

Input artifacts

```
SW: BINARY SW0
MINUS: ADDER_0
PLUS: ADDER +
MINUS: ADDER -
OUT := SW.BSW
```
Idea of the proposed method: cycle of refinement

1. Manual formalization
   - Textual specification
   - LTL specification

2. Generation
   - Generated model

3. Equivalence model-check
   - Example of differing behaviors
   - Manually created model

4a. Fix!
4b. Fix!

Output: log of faults in the model and specification ambiguities

Input artifacts

Combined use of formal methods for reliability assurance of software for safety-critical systems 28.06.2019 15 / 25
Example of an issue log

<table>
<thead>
<tr>
<th>Model</th>
<th>#</th>
<th>Description</th>
<th>Type of issue</th>
</tr>
</thead>
</table>
| Pulse | 1 | **Manual model:** initial output can be TRUE if input is TRUE. **Synthesized model:** the input on the first step is always FALSE.  
- Noticed when length in 0..1  
- **Action:** Fixed synthesis specifications to match the behavior of the manual implementation. | Specification ambiguity                  |
|       | 2 | **During a rising edge of input, which length/mode values should influence the duration of the pulse? The first or the second pair?**  
**Manual model:** the second pair of values.  
**Synthesized model:** no fixed policy.  
- Noticed when length in 0..1  
- **Action:** Fixed synthesis specifications to match the behavior of the manual implementation. | Specification ambiguity                  |
|       | 3 | **Manual model:** when PULSE_LENGTH is 0, modes 1..3 still give pulses of length 1 (while PULSE_REMAINING_TIME = 0, like in mode 0).  
**Synthesized model:** does not give pulses at all.  
- Noticed when length in 0..1  
- According to the Fortran source, the PULSE in modes 1...3 should not output anything. The manual implementation is incorrect. The case PULSE_LENGTH = 0 was not considered.  
- **Action:** Fixed synthesis specifications to match the behavior of the manual implementation. | Fault in the manual model                 |
### Example of an issue log

<table>
<thead>
<tr>
<th>Model</th>
<th>#</th>
<th>Description</th>
<th>Type of issue</th>
</tr>
</thead>
</table>
| Pulse | 1 | **Manual model:** initial output can be TRUE if input is TRUE. **Synthesized model:** the input on the first step is always FALSE.  
- Noticed when length in 0..1  
- **Action:** Fixed synthesis specifications to match the behavior of the manual implementation. | **Specification ambiguity** |
|       | 2 | **During a rising edge of input, which length/mode values should influence the duration of the pulse? The first or the second pair?**  
**Manual model:** the second pair of values.  
**Synthesized model:** no fixed policy.  
- Noticed when length in 0..1  
- **Action:** Fixed synthesis specifications to match the behavior of the manual implementation. | **Specification ambiguity** |
|       | 3 | **Manual model:** when PULSE_LENGTH is 0, modes 1..3 still give pulses of length 1 (while PULSE_REMAINING_TIME = 0, like in mode 0).  
**Synthesized model:** does not give pulses at all.  
- Noticed when length in 0..1  
- According to the Fortran source, the PULSE in modes 1...3 should not output anything. The manual implementation is incorrect. The case PULSE_LENGTH = 0 was not considered.  
- **Action:** Fixed synthesis specifications to match the behavior of the manual implementation. | **Fault in the manual model** |
A method was proposed to verify formal models that were manually created based on a natural language specification.

This problem is practically justified in nuclear I&C verification.

Experiments on four basic block models created in VTT Research Centre of Finland Ltd.

- Faults were found in two models.
- Specification ambiguities were found for three models.

The method is applicable for small models with several input and output variables.

Efficiency  Effectiveness  Automation

Contribution 2
• Contribution 1: methods to generate plant and controller models from specification

• Contribution 2: method to verify formal models created manually based on a natural language specification

• Contribution 3: testing method for closed-loop systems based on model checking
Why testing with formal models and model checking?

Plant model

Controller model
Why testing with formal models and model checking?

Plant model

Controller model

Combined use of formal methods for reliability assurance of software for safety-critical systems
Why testing with formal models and model checking?

Combined use of formal methods for reliability assurance of software for safety-critical systems

Plant model

Controller model

Closed-loop LTL specification

Computationally infeasible!
Why testing with formal models and model checking?

- **Plant model**
- **Controller model**
- **Closed-loop LTL specification**

Computationally infeasible!

Go back to **testing**, but let’s keep the benefits of formal approaches.
Proposed method applied on the elevator example
Proposed method applied on the elevator example

1. Extract user inputs from the plant model

Buttons in the car
Proposed method applied on the elevator example

1. Extract user inputs from the plant model

2. Generate tests as sequences of user inputs that cover all formal models

Buttons in the car

1. Extract user inputs from the plant model

2. Generate tests as sequences of user inputs that cover all formal models

Floor 3
Proposed method applied on the elevator example

1. Extract user inputs from the plant model

2. Generate tests as sequences of user inputs that cover all formal models

Floor 3 → Floor 2

1. Extract user inputs from the plant model

2. Generate tests as sequences of user inputs that cover all formal models

Floor 3 → Floor 2
Proposed method applied on the elevator example

1. Extract user inputs from the plant model

2. Generate tests as sequences of user inputs that cover all formal models

Buttons in the car

1. Extract user inputs from the plant model

2. Generate tests as sequences of user inputs that cover all formal models

Floor 3 → Floor 2 → Floor 1
1. Extract user inputs from the plant model

2. Generate tests as sequences of user inputs that cover all formal models

3. Execute these tests by running model checking on our specification

Both generation and execution are efficient!
• Area of applicability: **intractable computational complexity** of precise verification of closed-loop systems

• Test generation and execution by means of model checking were **unified** into a single **imprecise formal verification method**

• A family of (elevator) models was found, on which the proposed method:
  • works **faster** than model checking, and in particular bounded model checking (BMC)
  • **finds more specification violations** than BMC

---

**Contribution 3**

<table>
<thead>
<tr>
<th>Efficiency</th>
<th>Effectiveness</th>
<th>Automation</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️ ✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Conclusions
Conclusions

- Developed **eight methods** of reliability assurance for safety-critical control software
  - Implemented in Java and available on GitHub
  - Evaluated on practical problems

- Contributions and subgoals
  - Plant and controller generation methods → **increase automation of reliability assurance**
  - Verification of models from natural language specification → **increase effectiveness**
  - Testing in closed loop → **increase efficiency**

- The goal of the dissertation is achieved


Combined use of formal methods for reliability assurance of software for safety-critical systems

Igor Buzhinskii

Thank you for your attention!

Igor Buzhinskii

Combined use of formal methods for reliability assurance of software for safety-critical systems

2019, Espoo