Softwaretechnik / Software-Engineering

Lecture 17: Software Engineering Research

2015-07-16

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Schedule of the Block “Invited Talks”

- **12:15 - 12:17:39** — Introduction

- **12:17:53 - 12:55**
  - “The Wireless Fire Alarm System: Ensuring Conformance to Industrial Standards through Formal Verification”
  - **Sergio Feo Arenis**

- **12:55 - 13:05** — Break

- **13:05 - 13:30**
  - “Towards Successful Subcontracting for Software in Small to Medium-Sized Enterprises”
  - **Daniel Dietsch**

- **13:30 - 13:55**
  - “Traces, Interpolants, and Automata: a New Approach to Automatic Software Verification.”
  - **Dr. Jochen Hoenicke**

- **12:15 - 12:17:39** — Introduction

- **12:17:53 - 12:55**
  - “The Wireless Fire Alarm System: Ensuring Conformance to Industrial Standards through Formal Verification”
  - **Sergio Feo Arenis**

- **12:55 - 13:05** — Break

- **13:05 - 13:30**
  - “Towards Successful Subcontracting for Software in Small to Medium-Sized Enterprises”
  - **Daniel Dietsch**

- **13:30 - 13:55**
  - “Traces, Interpolants, and Automata: a New Approach to Automatic Software Verification.”
  - **Dr. Jochen Hoenicke**
Develop a wireless fire alarm system (safety critical).

Requires certification to international standards.

Small company with little to no experience with formal methods, but an acute need for product safety and quality.

Project duration: ca. 2 years.

Sergio Feo-Arenis (Uni. Freiburg)
Goals

Can formal methods handle development projects in the context of a small company (SME)? at what cost?

How to tackle requirements from industrial standards using formal methods?

What research ideas emerged from the project?

Sergio Feo-Arenis (Uni. Freiburg)

Wireless Fire Alarm System

SWT 2015 3 / 23
Scenario

- [ ] [ ] [ ] [ ]

- [ ] [ ] [ ] [ ]

- [ ] [ ] [ ] [ ]

- [ ] [ ] [ ] [ ]
Scenario

EN 54 Requirements:
- Detect and display communication failure in at least 300 ms
- Field even when there are other loops in the frequency

EN 54 Requirements:
- Display alarms clearly
- The display 10 seconds after the event and for at least 60 minutes for 10 simultaneous alarms
- Field even when there are other loops in the frequency
Challenges

Testing a design is difficult:

- There is a very large number of possible system configurations.
- Requires a prototype implementation.
- Controlling timing and radio communication environments requires costly procedures.
- The requirements assume an inherent nondeterminism.

Thus:

Verification could help.

Sergio Feo-Arenis (Uni. Freiburg)

Wireless Fire Alarm System

SWT 2015 5 / 23
General Risks

Development in a small company.

Development team of 3 people: 1 computer scientist, 1 programmer, 1 electrical engineer.

Underspecified standard requirements.

High cost of certification.

A failed certification attempt threatens the very existence of the company.

Market introduction deadlines have high priority.

Lack of structure in the software development process.

Weak documentation practices.

No familiarity with model-based development.

Sergio Feo-Arenis (Uni. Freiburg)

Wireless Fire Alarm System

SWT 2015 6 / 23
What to Verify: Requirements Formalization

EN-54 provides:
- High-level real-time requirements (hard to formalize).
- Test Procedures.
- Effort required: Months. It was necessary to negotiate ambiguities with the certification authority.

Chose duration calculus (DC) as formalism to generalize and capture the standard requirements based on test procedures.

The formalism was not familiar to developers or the certification authority.

Required developing a graphical means of communication between the stakeholders.

Sergio Feo-Arenis (Uni. Freiburg)
Wireless Fire Alarm System
SWT 2015 8 / 23
What to Verify: Requirements Formalization

Result of the DC formalization:
- Captured test procedures.
- Captured environment assumptions during tests (frequency jamming, simplifying assumptions).
- Generalized to cover all components in arbitrary system topologies.

Sergio Feo-Arenis (Uni. Freiburg)
Wireless Fire Alarm System
SWT 2015 10 / 23
What to Verify: Requirements Formalization

Result of the DC formalization:
- Captured test procedures.
- Captured environment assumptions during tests (frequency jamming, simplifying assumptions).
- Generalized to cover all components in arbitrary system topologies.

In total:
- 6 (quantified) observables
- 7 (quantified) testable DC formulae

Sergio Feo-Arenis (Uni. Freiburg)
Wireless Fire Alarm System
SWT 2015 10 / 23
Modeling: Monitoring Function

Decomposition gives way to additional proof obligations:
- No interference between networks (by design)
- No collisions (TDMA)
- Guard time analysis

Topology subsumption: Verifying a maximal subnetwork is enough.
Modeling: Monitoring Function

Decomposition gives way to additional proof obligations:

- No interference between networks (by design).
- No collisions (TDMA).
- Guard time analysis.
- Topology subsumption: Verifying a maximal subnetwork is enough.

To make models tractable, we require optimization:

- Each component has an individual clock.
- Quasi-equal clock reduction.
- Support plug-in models: Separate environment and design.

Sergio Feo-Arenis (Uni. Freiburg)

Wireless Fire Alarm System

SWT 2015 12 / 23

Modeling: Sensor Failures

Modeled as timed automata networks with UPPAL.
Modeling: Sensor Failures

Verification: Monitoring Function

Other model components:
- Auxiliary components
- Inner network
- Stamps

Sergio Feo-Arenis (Uni. Freiburg)
### Verification: Monitoring Function

- Other model components:
  - Auxiliary automata: Master, Central clock, Monitor
  - Inner network: 10 Repeaters

- Found 2 flaws:
  1. Timing was off by 1 tic
  2. Frequency intrusion

A revised design was successfully verified:

- Sensors as slaves
- Repeaters as slaves

<table>
<thead>
<tr>
<th>Query seconds</th>
<th>MB States</th>
<th>6M</th>
<th>0.15M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection</td>
<td>36,070.78</td>
<td>3,419.00</td>
<td>190M</td>
</tr>
<tr>
<td>No Spurious</td>
<td>97.44</td>
<td>44.29</td>
<td>0.6M</td>
</tr>
<tr>
<td>No LZ-Collision</td>
<td>12,895.17</td>
<td>2,343.00</td>
<td>68M</td>
</tr>
<tr>
<td>Detection Possible</td>
<td>10,205.13</td>
<td>557.00</td>
<td>26M</td>
</tr>
</tbody>
</table>

Verification is scalable for real world problems (!). But additional effort is required.

Sergio Feo-Arenis (Uni. Freiburg)

Wireless Fire Alarm System
### Verification: Alarm Function

For single, explicit topologies: Timed automata / UPPAAL.

<table>
<thead>
<tr>
<th></th>
<th>OneAlarm</th>
<th>TwoAlarms</th>
<th>TenAlarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ids</td>
<td>1.4 ± 1</td>
<td>0.5 ± 1</td>
<td>17.1 ± 6</td>
</tr>
<tr>
<td>seconds</td>
<td>38.3 ± 1</td>
<td>24.1 ± 2</td>
<td>179.1 ± 6</td>
</tr>
<tr>
<td>MB</td>
<td>36.4 ± 1</td>
<td>19,528 ± 4</td>
<td>419 ± 6</td>
</tr>
<tr>
<td>States</td>
<td>38. ± 1</td>
<td>17.3 ± 6</td>
<td>412 ± 6</td>
</tr>
</tbody>
</table>

Limited Collision

<table>
<thead>
<tr>
<th></th>
<th>OneAlarm</th>
<th>TwoAlarms</th>
<th>TenAlarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ids</td>
<td>1.4 ± 1</td>
<td>0.5 ± 1</td>
<td>17.1 ± 6</td>
</tr>
<tr>
<td>seconds</td>
<td>38.3 ± 1</td>
<td>24.1 ± 2</td>
<td>179.1 ± 6</td>
</tr>
<tr>
<td>MB</td>
<td>36.4 ± 1</td>
<td>19,528 ± 4</td>
<td>419 ± 6</td>
</tr>
<tr>
<td>States</td>
<td>38. ± 1</td>
<td>17.3 ± 6</td>
<td>412 ± 6</td>
</tr>
</tbody>
</table>

Checking one topology is feasible, but the procedure does not scale for full verification (more than 10^126 possible topologies).

[Parameterized Verification of Aggregation Protocols]

Models are still useful for simulation: extracted expected alarm times for different scenarios.

Sergio Feo-Arenis (Uni. Freiburg)

Wireless Fire Alarm System

SWT 2015 18 / 23
Verification: Alarm Function

For increased confidence: Does the collision resolution algorithm guarantee non-starvation?

Sergio Feo-Arenis (Uni. Freiburg)

Wireless Fire Alarm System

SWT 2015 19 / 23

Verification: Alarm Function

For increased confidence: Does the collision resolution algorithm guarantee non-starvation? Created an untimed model in PROMELA / SPIN.

$N$: number of colliding components. $I$: set of IDs that may participate in the collision.

Check all possible $N$-collision scenarios: vary IDs and timing.
Verification: Alarm Function

For increased confidence: Does the collision resolution algorithm guarantee non-starvation? Created an untimed model in PROMELA / SPIN.

- $N$: number of colliding components.
- $I$: set of IDs that may participate in the collision.

Check all possible $N$-collision scenarios: vary IDs and timing.

Results:
- Reproduced the hidden terminal problem.
- For $N = 2$: found a problem with IDs 0 and 128.
- For $N = \{3 \ldots 10\}$: still not scaling to all IDs, used sampling (31744).

<table>
<thead>
<tr>
<th>N</th>
<th>I</th>
<th>States</th>
<th>H</th>
<th>L</th>
<th>Rnd average</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>128</td>
<td>49</td>
<td>3,393</td>
<td>4,271</td>
<td>4,465</td>
</tr>
<tr>
<td>3</td>
<td>6,390</td>
<td>1,610</td>
<td>10,685</td>
<td>10,439,545</td>
<td>11,534</td>
</tr>
<tr>
<td>4</td>
<td>9,994</td>
<td>1,235,970</td>
<td>6,242,610</td>
<td>11,268,368</td>
<td>11,534</td>
</tr>
<tr>
<td>5</td>
<td>9,763,809</td>
<td>255</td>
<td>6,390</td>
<td>4,138</td>
<td>4,138</td>
</tr>
</tbody>
</table>

Sergio Feo-Arenis (Uni. Freiburg)

Lessons Learned

Generalized test procedures are useful for verification:

1. Developers are already used to producing test specifications.
2. Thus, they are motivated to perform verification.
3. Generalized tests are already used to produce the standard.

For increased confidence: Algorithms.

Lessons Learned
Lessons Learned

Generalized test procedures are useful for verification:
Developers are already used to producing test specifications.
Thus: are cost-effective for increasing confidence.

Models are useful:
For validation.
As documentation.
But still not very accessible for developers.

Formal verification shows potential to relieve the effort of testing.

Sergio Feo-Arenis (Uni. Freiburg)

Conclusions

Formal methods are able to handle typical industrial scenarios (but require expert knowledge).

The customers are confident early in the process that certification tests will be passed.

Implementation is easier when based on a verified design.
Other requirements can be simply tested.

Still expensive: Almost as expensive as the certification test itself.
Additional value: Formal methods not only improve confidence but help structure development processes.

Difficult technology transfer: SMEs prefer to scale out instead of up.

Sergio Feo-Arenis (Uni. Freiburg)
Outlook

- Check whether the source code of the implementation corresponds to the design models. Interrupt-based implementations are hard to verify.
- Use the models to perform model-based testing.
- Investigate reuse strategies (new features, product lines).
Towards Successful Sub contracting for Software in Small to Medium-Sized Enterprises

Bernd Westphal, Daniel Dietsch, Sergio Feo-Arenis, Andreas Podelski, Louis Pahlow, Jochen Morsbach, Barbara Sommer, Anke Fuchs, Christine Meierhöfer

1 Albert-Ludwigs-Universität Freiburg, Germany
2 Universität des Saarlandes, Saarbrücken, Germany
3 Universität Mannheim, Germany

Outline

Introduction
- What is sub-contracting for software?
- When is it successful?
- Why is it often not successful?

The Salomo Approach:
- Overview
- Checkable Requirements, Checking Tool
- Regulations in the Contract

Related Work

Conclusion and Further Work
Successful Sub contracting for Software in SMEs

SME A

Customer

Contractor

SME B

Software
Successful Sub contracting for Software in S Es

Su contracting for Software in S Es in Realit
There are three main sources of disputes (and thus uncertainty):

- misunderstandings in the requirements,
Bringing Software-related Disputes to Court is generally highly unattractive for SME:

(i) a court ruling takes time, thus further delays the project,

(ii) a court ruling incurs costs,
Bringing Software-related Disputes to Court is generally highly unattractive for SME:

(i) a court ruling takes time, thus further delays the project,
(ii) a court ruling incurs costs,
(iii) it is uncertain whether the necessary compensation can be achieved,
(iv) a court only decides over the rights and duties of each party, no suggestion how to use the decision to achieve project success,
Bringing Software-related Disputes to Court is generally highly unattractive for SME:

(i) a court ruling takes time, thus further delays the project,
(ii) a court ruling incurs costs,
(iii) it is uncertain whether the necessary compensation can be achieved,
(iv) a court only decides over the rights and duties of each party, no suggestion how to use the decision to achieve project success,
(v) mutual trust between the former partners is hampered, already achieved project progress may be lost.

In addition, there is a high uncertainty about the outcome:

- given unclear requirements, an appointed expert witness may confirm either interpretation.
There are three main sources of disputes (and thus uncertainty):

- misunderstandings in the requirements,
- misunderstandings or (under-regulations) of acceptance testing procedure,
There are three main sources of disputes (and thus uncertainty):

- misunderstandings in the requirements,
- misunderstandings or (under-regulations) of acceptance testing procedure,
- misunderstandings of regulations of the contract.

Many SMEs conclude: subcontracting for software is too risky due to these three main sources of uncertainty.
• **(Legal) certainty** is crucial for subcontracting between SMEs: Outcomes of possible court judgements need to be as clear as possible.

• To achieve legal certainty, we need
  
  (a) **clear and precise requirements**, they avoid the 1st source of uncertainty.
  
  (b) **clear and precise acceptance testing procedures**, they avoid the 2nd source of uncertainty.
  
  (c) **standardised legal contracts** which integrate (a) and (b), they avoid the 3rd source of uncertainty.

  The contract allows a judge to decide on (a) and (b), and thus increases legal certainty.

---

**Outline**

• Introduction
  
  • What is sub-contracting for software?
  
  • When is it successful?
  
  • Why is it often not successful?
  
  ◮ The Salomo Approach:
    
    • Overview
    
    • Checkable Requirements, Checking Tool
    
    • Regulations in the Contract
  
  • Related Work
  
  • Conclusion and Further Work
Towards Legal Certainty

Ingredients:
- new notion: checkable requirement,
- new notion: checking tool.
- a new, modular software development contract,

The modular contract

assumes: a subset of requirements is designated as checkable requirements,
includes: the checkable requirements in machine-readable form,
codifies: agreement that outcome of corresponding checking tool is — with few and exactly specified exceptions — binding for both parties,
provides: legal certainty.

Checkable Specification/Requirement, Checking Tool

- A checkable specification is a pair \((\varphi, T)\)
  comprising a program property \(\varphi\) and a backend \(T\).

- A backend maps a program \(p\) and a program property \(\varphi\)
to a result \(T(p, \varphi) \in \{Yes, No, Unknown\}\) such that the result is
  - Yes only if the program has the property,
  - No only if the program does not have the property.
A **checkable specification** is a pair \((\varphi, T)\) comprising a **program property** \(\varphi\) and a **backend** \(T\).

A **backend** maps a program \(p\) and a program property \(\varphi\) to a result \(T(p, \varphi) \in \{\text{Yes}, \text{No}, \text{Unknown}\}\) such that the result is
- **Yes** only if the program has the property,
- **No** only if the program does not have the property.

A **checking tool** maps a set of checkable specifications
\[
\Phi = \{(\varphi_1, T_1), \ldots, (\varphi_n, T_n)\}, \ n \in \mathbb{N}_0,
\]
to a **checking tool result**
\[
\{(\varphi_1, s_1), \ldots, (\varphi_n, s_n)\}, s_i \in \{\text{Yes}, \text{No}, \text{Unknown}\}.
\]

A requirement is called **checkable requirement** if and only if a checkable specification can (mechanically) be derived from it.
Backend Examples

- “The Program Compiles”: wrapper applies compiler and yields
  - Yes, compiler $C$ in version $V$ produces a non-empty executable.
  - No, otherwise.

- “Test Coverage”: wrapper applies unit-tests
  - Yes, normal termination of unit tests indicates 100% branch coverage,
  - No, normal termination and branch coverage below 100%,
  - Unknown, otherwise.
Backend Examples

- **"The Program Compiles"**: wrapper applies compiler and yields
  - Yes, compiler $C$ in version $V$ produces a non-empty executable.
  - No, otherwise.

- **"Test Coverage"**: wrapper applies unit-tests
  - Yes, normal termination of unit tests indicates 100% branch coverage,
  - No, normal termination and branch coverage below 100%,
  - Unknown, otherwise.

- **"Absence of Generic Errors"**: wrapper applies, e.g., Frama-C
  - Yes, all assertions related to safe memory access hold or not tried,
  - No, at least one assertion has status surely_invalid, and
  - Unknown otherwise.

- **"Invariant Satisfied"**: wrapper applies, e.g., VCC
  - Yes, verifier output indicates invariant proven; Unknown, otherwise.
Backend Examples

• “The Program Compiles”: wrapper applies compiler and yields
  • Yes, compiler C in version V produces a non-empty executable.
  • No, otherwise.

• “Test Coverage”: wrapper applies unit-tests
  • Yes, normal termination of unit tests indicates 100% branch coverage,
  • No, normal termination and branch coverage below 100%,
  • Unknown, otherwise.

• “Absence of Generic Errors”: wrapper applies, e.g., Frama-C
  • Yes, all assertions related to safe memory access hold or not tried,
  • No, at least one assertion has status surely_invalid, and
  • Unknown otherwise.

• “Invariant Satisfied”: wrapper applies, e.g., VCC
  • Yes, verifier output indicates invariant proven; Unknown, otherwise.

• “Certification”: expert reviews of programs

Regulations in the Contract

• The modular software development contract
  • consists of a framework contract, referred to by individual contract,
  • customisation by several contractual modules.
Regulations in the Contract

- The modular software development contract
  - consists of a framework contract, referred to by individual contract,
  - customisation by several contractual modules.

- The acceptance checking procedure is regulated in two clauses:
  (i) checkable requirements tested with and only with checking tool.
      Exit option: if
      - backend is evidently erroneous, or
      - the parties agree to consider the result erroneous, or
      - there is an "Unknown" among only "Yes"s and "Unknown"s,
        then the clause for other requirements applies.

  (ii) testing procedure for other requirements determined by customer.

Outline

- Introduction
  - What is sub-contracting for software?
  - When is it succesful?
  - Why is it ofen not successful?

- The Salomo Approach:
  - Overview
  - Checkable Requirements, Checking Tool
  - Regulations in the Contract

  ▶ Related Work

- Conclusion and Further Work
Related Work

- (Berenbach, Lo & Sherman, 2010)
  Scope limited to the time after the contract has been awarded, limited discussion regarding contract compliance check.

- (Governatori, Milosevic, & Sadiq, 2006) — formalise contract conditions
  Use FCL to formalise requirements business rules and tools which decide compliance as acceptance checking procedure.

- (Breaux, Antón, Spafford, 2009) — delegation
  We consider top-level obligations and verification sets without delegation.

- (Fanmuy, Fraga & Lloréns, 2012) — requirements verification
  Use requirements verification as acceptance checking procedure if creation of a requirements document is subject of a contract.

Conclusion and Further Work

- We tackle a main challenge of contracting for software: legal uncertainty.
- We outline a possible approach to resolve three reasons of uncertainty: a modular legal contract codifies the mutual agreement that checkable requirements are verified by checking tool exclusively.
- Both, contractor and customer have strong interest in obtaining positive checking results since positive results mean certainty.
- Our contract is well-suited for a gradual introduction of formal methods — any backend is supported as long as both parties agree.
- Formal methods effort promises increased confidence in software quality.

Further work:
- legally support traceability, change-requests.
- consider a concept of delegation similar to (Breaux et al., 2009),
- provide more backends.
Thanks.
Software Verification

- prove or disprove that a given program satisfies a given specification

- problem is undecidable [Turing, 1936]
Example

ℓ₀: assume p ≠ 0;

ℓ₁: while (n ≥ 0)

ℓ₂: assert p ≠ 0; if (n == 0)

ℓ₃: p := 0;

ℓ₄: n--;

ℓ₅: 

pseudocode

ℓ₀
ℓ₁
ℓ₂
ℓ₃
ℓ₄
ℓ₅

err

p ≠ 0
n ≥ 0
n == 0
p := 0
n --

control flow graph
Example

\[ \ell_0: \text{assume } p \neq 0; \]

\[ \ell_1: \text{while}(n \geq 0) \{
\]

\[ \ell_2: \text{assert } p \neq 0; \text{if}(n == 0) \{
\]

\[ \ell_3: p := 0; \}

\[ \ell_4: n--; \}

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]

\[ \]
1. take trace $\pi$
2. consider trace as program $P$
3. analyze correctness of $P$

### Pseudocode of $P$

1. assume $p \neq 0$;
2. assume $n \geq 0$;
3. assert $p \neq 0$;

$p \neq 0$

$n \geq 0$

$p = 0$

true

$p \neq 0$

false

✓✓✓
1. Take trace π
2. Consider trace as program P
3. Analyze correctness of P
4. Generalize program P
   \( \Delta \) add transitions

\[ \{ p \neq 0 \} \]
\[ n--; \{ p \neq 0 \} \]

is valid Hoare triple

\[ \{ p \neq 0 \} \]
\[ n \neq 0--; \{ p \neq 0 \} \]

is valid Hoare triple

\[ p \neq 0 \]
\[ n \geq 0 \]
\[ p = 0 \]
\[ true \]
\[ p \neq 0 \]
\[ p \neq 0 \]
\[ false \]

✓✓✓
1. take trace \( \pi \)
2. consider trace as program \( P \)
3. analyze correctness of \( P \)
4. generalize program \( P \)

\[ \begin{align*}
\text{add transitions } & \{ p \neq 0 \} \\
& \text{n--} \\
\end{align*} \]

is valid Hoare triple

\[ \{ p \neq 0 \} \quad \text{n != 0} \quad \{ p \neq 0 \} \]

is valid Hoare triple

\[ \{ p \neq 0 \} \quad \text{n >= 0} \quad \{ p \neq 0 \} \]

is valid Hoare triple

\[ \text{p != 0} \quad \text{n >= 0} \quad \text{p == 0} \]

true

\[ \text{p} \neq 0 \quad \text{p} \neq 0 \]

false
1. take trace π
2. consider trace as program P
3. analyze correctness or
4. generalize program P

◮ add transitions
◮ merge locations

p \neq 0
p = 0
true
false

all \( \{ p := 0 \} \)

✓✓✓
A program defines a language over the alphabet of statements.
New View on Programs

“A program defines a language over the alphabet of statements.”

◮ Set of statements: alphabet of formal language
   - e.g., Σ = {p ≠ 0, n ≥ 0, n == 0, p := 0, n ≠ 0, p == 0, n--, n < 0, ...}

◮ Control flow graph: automaton over the alphabet of statements

◮ Error location: accepting state of this automaton
"A program defines a language over the alphabet of statements."

- Set of statements: alphabet of formal language
  - \[ \Sigma = \{ p \neq 0, n \geq 0, n = 0, p := 0, n \neq 0, p = 0, n --, n < 0 \} \]

- Control flow graph: automaton over the alphabet of statements
- Error location: accepting state of this automaton
- Error trace of program: word accepted by this automaton
1. take trace \( \pi \)
2. \( p \neq 0 \)
3. \( n \geq 0 \)
4. \( n = 0 \)
5. \( p := 0 \)
6. \( n-- \)
7. \( n \geq 0 \)
8. \( p = 0 \)

2. consider trace as program \( P \)
1. take trace \( \pi \)
2. consider trace as program \( P \)
3. analyze correctness or

\[
p \neq 0 \\
n \geq 0 \\
n = 0 \\
p := 0 \\n\]

\[
n -- \\
n \geq 0 \\
n = 0 \\
true \\
true \\
true \\
n \neq -1 \\
false \\
false \\
true \\
true \\
true \\
\]

4. generalize program \( P \)
   ▶ add transitions
   ▶ merge locations

\( q_0 \)
\( q_1 \)
\( q_2 \)
\( q_3 \)

\[
\{ n -- \} \forall \{ n -- \} \\
\]
Verification Algorithm

program $P$

"$P$ is correct"

$P$ is incorrect

$L(P) \subseteq L(P_1) \cup \cdots \cup L(P_n)$

is $\pi$ feasible?

no pick new error trace

$\pi$ no construct infeasibility proof for $\pi$

construct generalized program $P_i$

yes yes

Verification Algorithm
Verification Algorithm

Program $P$

"$P$ is correct"

"$P$ is incorrect"

$L(P) \subseteq L(P_1) \cup \cdots \cup L(P_n)$

Is $\pi$ feasible?

No: pick new error trace

Yes: yes

Pick an error trace $\pi$, is $\pi$ feasible?

No: construct infeasibility proof for $\pi$

Yes: $P_i$ is correct, go to line 24

"$P$ is incorrect"
Verification Algorithm

Program $P$

- $P$ is correct
- $P$ is incorrect

Let $L(P) \subseteq L(P_1) \cup \ldots \cup L(P_n)$

Is $\pi$ feasible?

- No
  - Pick new error trace
- Yes
  - Construct infeasibility proof for $\pi$
  - Construct generalized program $P_i$

Interprocedural/Recursive Programs
Recursive Programs - Challenge 1: Control Flow

procedure m(x) returns (res)

ℓ₀:
if x > 100
ℓ₁:
res := x - 10
else
ℓ₂:
x := x + 11
ℓ₃:
call m
ℓ₄:
x := res
m
ℓ₅:
call m
ℓ₆:
res := res
m
ℓ₇:
assert (x ≤ 101 -> res = 91)
return m
Recursive Programs - Challenge 2: Local Annotations

What is an annotation for an interprocedural execution?

◮ state with a stack?
⇝ locality of annotation is lost

\[ x_p = 0 \]
\[ x_p = 0 \]
\[ x = 0 \]
\[ x_p = 0 \]
\[ x_p = -1 \]
\[ x_p = 0 \]
\[ x_p = -1 \]
\[ x = 1 \]
\[ x_p = 0 \]
\[ x_p = -1 \]
\[ x_p = 0 \]
\[ x_p = 0 \]
\[ x_p = 0 \]
\[ x_p = -1 \]
\[ x_p = 0 \]
\[ x_p = 0 \]
\[ false \]
\[ x_p := 0 \]
\[ call p \]
\[ x_p := x - 1 \]
\[ call p \]
\[ res := x \]
\[ return \]
\[ res := res_p - x_p \]
\[ return \]
\[ res_p < x_p \]
Recursive Programs - Challenge 2: Local Annotations

What is an annotation for an interprocedural execution?

Idea: "Nested Interpolants"

Define sequence of state assertions with respect to nested trace.

```
x = 0
call p
x = x - 1
call p
res = x
return
```

```
x = x
p = x - 1
return p
```

```
true
x = 0
true
x = x - 1
true
res = x
res \geq x
res \geq x
false
```

Define ternary post operator for return statements

```
post( res = x, x = x - 1, return p ) \subseteq res \geq x
```

Local state of caller before call
Local state of callee before return
Local state of caller after return

Termination Analysis
Termination Analysis

Challenge 1: counterexample to termination is infinite execution

Solution: consider infinite traces, use $\omega$-words and Büchi automata
Termination Analysis

◮ Challenge 1: counterexample to termination is infinite execution

Solution: consider infinite traces, use \( \omega \)-words and Büchi automata

◮ Challenge 2: An infinite trace may not have any execution although each finite prefix has an execution. E.g.,

\[
\begin{align*}
\text{while} & (x > 0) \\
\{ & x--; \\
& \text{...}
\end{align*}
\]

Solution: ranking functions (here: \( f(x) = x \) )

Ranking Function (for a Loop)
Function from program states to well-founded domain such that value is decreasing while executing the loop body.
Proof by contradiction for the absence of infinite executions.
Example: Bubble Sort

Program: `sort(int i, int a[])`

1. `while (i > 0)`
2. `int j := 1`  
3. `while (j < i)`
4. `if (a[j] > a[i])`  
5. `swap(a, i, j)`  
6. `j++`
7. `i--`

Example: Bubble Sort

Program: `sort(int i)`

1. `while (i > 0)`
2. `int j := 1`  
3. `while (j < i)`
4. `j++`
5. `i--`
Example: Bubble Sort

```plaintext
class BubbleSort {
    func sort(arr: [Int]) -> [Int] {
        var n = arr.count
        while n > 0 {
            var j = 0
            while j < n - 1 {
                if arr[j] > arr[j + 1] {
                    arr.swapAt(j, j + 1)
                }
                j += 1
            }
            n -= 1
        }
        return arr
    }
}
```

quadratic ranking function:

```
f(i, j) = i^2 - j
```

lexicographic ranking function:

```
f(i, j) = (i, i - j)
```
\[ \omega = (\text{Inner} \star \text{Outer}) \]
From $\omega$-Trace to Terminating Program – Example

input: ultimately periodic trace

\[ i > 0 \]

\[ j := 1 \]

\( \begin{cases} 
j < i \\
 j++ 
\end{cases} \)

\[ \omega, \]
1. Consider $\omega$-trace as a program with a single while loop

2. Synthesize ranking function $f(i, j) = i - j$

3. Compute rank certificate $\ell_1$ oldrnk = $\infty$, $\ell_2$ oldrnk = $\infty$, $\ell_3$ $i - j \prec oldrnk$, $\ell_4$ $i - j \leq oldrnk \land i - j \geq 0$
From ω-Trace to Terminating Program – Example

input: ultimately periodic trace

1. consider ω-trace as program with single while loop

2. synthesize ranking function

3. compute rank certificate

4. add additional transitions

Generalization of Program with Rank Certificate

◮ Case 1:

Hoare triple

\{ state assertion 1 \}

stmt

\{ state assertion 2 \}
Generalization of Program with Rank Certificate

Case 1: $q_1$ not accepting Hoare triple 
\{ state assertion 1 \} stmt 
\{ state assertion 2 \}

automaton transition 
$q_1$ state assertion 1 $q_2$ state assertion 2 stmt

Case 2: $q_1$ accepting Hoare triple 
\{ state assertion 1 \} oldrnk:=f(x) stmt 
\{ state assertion 2 \}

automaton transition 
$q_1$ state assertion 1 $q_2$ state assertion 2 oldrnk:=f(x) stmt

Implemented in Ultimate B"uchi Automizer
http://ultimate.informatik.uni-freiburg.de/BuchiAutomizer/
Implemented in Ultimate Büchi Automizer
http://ultimate.informatik.uni-freiburg.de/BuchiAutomizer/

For synthesis of ranking functions for single traces we use the tool:
Ultimate LassoRanker
http://ultimate.informatik.uni-freiburg.de/LassoRanker/
developed together with Jan Leike

Programs with procedures and recursion? Büchi Nested Word Automata!
Future Work

Future Work