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

Context

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.

Goals

- Can formal methods handle development projects in the context of a small company (SME)? at which cost?
- How to tackle requirements from industrial standards using formal methods?
- What research ideas emerged from the project?

Sergio Feo Arenis (Uni. Freiburg)
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.

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.

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 certificate authority.

Required developing a graphical means of communication between the stakeholders.

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.
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

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.

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.

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.

Verification: Monitoring Function

Other model components:

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

Found 2 flaws:

- Timing was off by 1 tic
- Frequency intrusion

A revised design was successfully verified:

- Sensors as slaves
- Repeaters as slaves

<table>
<thead>
<tr>
<th>Detection</th>
<th>Seconds</th>
<th>MB</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>36,070.78</td>
<td>3,419.00</td>
<td>190M</td>
<td></td>
</tr>
<tr>
<td>231.84</td>
<td>230.59</td>
<td>6M</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No Spurious</th>
<th>Seconds</th>
<th>MB</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>97.44</td>
<td>44.29</td>
<td>0.6M</td>
<td></td>
</tr>
<tr>
<td>3.94</td>
<td>10.14</td>
<td>0.15M</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No LZ-Collision</th>
<th>Seconds</th>
<th>MB</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>12,895.17</td>
<td>2,343.00</td>
<td>68M</td>
<td></td>
</tr>
<tr>
<td>368.58</td>
<td>250.91</td>
<td>9.6M</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Detection Possible</th>
<th>Seconds</th>
<th>MB</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,205.13</td>
<td>557.00</td>
<td>26M</td>
<td></td>
</tr>
<tr>
<td>38.21</td>
<td>55.67</td>
<td>1.2M</td>
<td></td>
</tr>
</tbody>
</table>

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

For single, explicit topologies: Timed automata / UPPAAL.

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 \( N = \{3, 4, ..., 10\} \): still not scaling to all IDs, used sampling (31744).

<table>
<thead>
<tr>
<th>( I )</th>
<th>( N ) sec.</th>
<th>MB</th>
<th>States</th>
<th>( H ) states</th>
<th>( L ) states</th>
<th>( Rnd ) states</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>49</td>
<td>1,610</td>
<td>1,235,970</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3,393</td>
<td>6,390</td>
<td>6,242,610</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>49</td>
<td>1,610</td>
<td>1,235,970</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>4,271</td>
<td>10,685</td>
<td>10,439,545</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>49</td>
<td>1,610</td>
<td>1,235,970</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>4,465</td>
<td>11,534</td>
<td>11,268,368</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

average

- 4,138
- 9,994
- 9,763,809

Sergio Feo-Arenis (Uni. Freiburg)

Wireless Fire Alarm System

SWT 2015 19 / 23

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)

Wireless Fire Alarm System

SWT 2015 20 / 23

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 helps structure development processes.
- Difficult technology transfer: SMEs prefer to scale out instead of up.

Sergio Feo-Arenis (Uni. Freiburg)

Wireless Fire Alarm System

SWT 2015 21 / 23

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).

Sergio Feo-Arenis (Uni. Freiburg)

Wireless Fire Alarm System

SWT 2015 22 / 23
Towards Successful Sub contracting for Software in Small to Medium-Sized Enterprises

RELAW Workshop, 2012-09-25

Bernd Westphal
1
Daniel Dietsch
1
Sergio Feo-Arenis
1
Andreas Podelski
1
Louis Pahlow
2
Jochen Morsbach
3
Barbara Sommer
3
Anke Fuchs
3
Christine Meierhofer
3

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 SM Es

There are three main sources of disputes (and thus uncertainty):

• misunderstandings in the requirements,

(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,
(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,
• misunderstandings of regulations of the contract.

Many SMEs conclude: subcontracting for software is too risky due to these three main sources of uncertainty.

– 0 – 2012-09-25 – Sreality – 7/17

**Observation**

• (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.

– 0 – 2012-09-25 – main – 8/17

**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

– 0 – 2012-09-25 – main – 9/17

Towards (Legal) Certainty

Ingredients:

• new notion: checkable requirement,
• new notion: checking tool.

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.

– 0 – 2012-09-25 – main – 10/17

**Checkable Specification/Requirement, Checking Tool**

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

• A backend maps a program \(p\) and a program property \(\phi\) to a result \(T(p, \phi) \in \{Yes, No, Unknown\}\) such that the result is

• Yes only if the program has the property,
• No only if the program does not have the property.

– 0 – 2012-09-25 – main – 11/17
Because the checking tool result \( \Phi \) maps a set of checkable specifications to a result such that the result is a pair \( \phi, T \) comprising a checkable specification \( \phi \) and a program property \( T \) that the program does not have the property, has the program only if the property.

\( \Phi = \{ \Delta \mid \vec{N} \in \{0, \ldots, n\} \} \) maps a set of checkable specifications to a result such that the result is a pair \( \phi, T \) comprising a checkable specification \( \phi \) and a program property \( T \) that the program does not have the property, has the program only if the property.

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

Unknown

\( \{0, \ldots, n\} \) contains at least one assertion has status No, and invalid surely, at least one assertion has status Yes. 

backend Examples

C heckab le S pecifi cation/Req uirem ent, C hecking T ool

produces a non-empty executable.

\( \overline{V_C} \) is a pair \( \phi, T \) comprising a checkable specification \( \phi \) and a program property \( T \) that the program does not have the property, has the program only if the property.

If \( N \in \{0, \ldots, n\} \) and if a checkable specification \( \phi \) and a program property \( T \) that the program does not have the property, has the program only if the property.

\( \{\overline{pi}, \phi \} \mid \overline{T} \) is a pair \( \phi, T \) comprising a checkable specification \( \phi \) and a program property \( T \) that the program does not have the property, has the program only if the property.

\( \{0, \ldots, n\} \) contains at least one assertion has status No, and invalid surely, at least one assertion has status Yes.

backend Examples

C heckab le S pecifi cation/Req uirem ent, C hecking T ool

produces a non-empty executable.

\( \overline{V_C} \) is a pair \( \phi, T \) comprising a checkable specification \( \phi \) and a program property \( T \) that the program does not have the property, has the program only if the property.

If \( N \in \{0, \ldots, n\} \) and if a checkable specification \( \phi \) and a program property \( T \) that the program does not have the property, has the program only if the property.

\( \{\overline{pi}, \phi \} \mid \overline{T} \) is a pair \( \phi, T \) comprising a checkable specification \( \phi \) and a program property \( T \) that the program does not have the property, has the program only if the property.
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.

  • 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.

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.
Traces, Interpolants, and Automata: a New Approach to Automatic Software Verification

Jochen Hoenicke
University of Freiburg
joint work with Andreas Podelski and Matthias Heizmann
16 July 2015

Software Verification
prove or disprove that a given program satisfies a given specification

Example
\[\ell_0: \text{assume } p \neq 0; \]
\[\ell_1: \text{while}(n \geq 0) \{
  \ell_2: \text{assert } p \neq 0;
  \text{if}(n == 0) \{
    \ell_3: p := 0;
  \}
  \ell_4: n--;
\}\]

pseudocode

keyboard

control flow graph
Example

1: assume \( p \neq 0 \);

2: while \( n \geq 0 \)

3: assert \( p \neq 0 \);

4: if \( n = 0 \)

5: \( p := 0 \);

6: \( n--; \)
1. take trace $\pi$
2. consider trace as program $P$
3. analyze correctness or $P$
4. generalize program $P$
   $\triangleleft$ 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
   $\{ p \neq 0 \}$
   $n \geq 0$
   $\{ p \neq 0 \}$
   is valid Hoare triple
   $p \neq 0$
   $n \geq 0$
   $p = 0$
   true
   $p \neq 0$
   $p \neq 0$
   false
   ✓✓✓
   all
   $\{ p := 0 \}$

New View on Programs
"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 \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. consider trace as program \( P \)

3. analyze correctness of \( P \)

4. generalize program \( P \)

- add transitions
- merge locations

\[ q_0 \quad q_1 \quad q_2 \quad q_3 \]

\[ \ell_0 \quad \ell_1 \quad \ell_2 \quad \ell_3 \quad \ell_4 \quad \ell_5 \quad \ell_{\text{err}} \]

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

\[ n \neq 0 \quad p \neq 0 \quad n = 0 \quad p := 0 \quad n \downarrow \quad n < 0 \]

\[ P \subseteq P_1 \cup P_2 \]
Verification Algorithm

program $P$

"$P$ is correct"

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

is $\pi$ feasible?

no pick new error trace $\pi$

no construct infeasibility proof for $\pi$

construct generalized program $P_i$

yes yes
What is an annotation for an interprocedural execution?

Idea: "Nested Interpolants"

Define ternary post operator for return statements

Recursive Programs - Challenge 2: Local Annotations

Locality of annotation is lost

Sequence of state assertions is not a proof

What is an annotation for an interprocedural execution?

Recursive Programs - Challenge 1: Control Flow

Control flow graph

McCarthy 91 function

procedure m(x) returns (res)

if x > 100

else

assert (x ≤ 101 → res = 91)

end if

return m

call m

res := res

return m

call m

res := res

...
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., $(x > 0 \ x--)$

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]) swap(a, i, j)
  \[5\]: j++
  \[6\]: i--

Example: Bubble Sort

program sort(int i)
  \[1\]: while (i > 0)
  \[2\]: int j := 1
  \[3\]: while (j < i)
  \[4\]: j++
  \[5\]: i--
  \[6\]: i > 0
  \[7\]: j := 1
  \[8\]: j < i
  \[9\]: j++
  \[10\]: j >= i
  \[11\]: i--
Example: Bubble Sort

```c
program sort(int i)
  ℓ1: while (i>0)
    ℓ2: int j:=1
    ℓ3: while(j<i)
      ℓ4: j++
      ℓ5: i--
```

quadratic ranking function:

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

lexicographic ranking function:

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

From ω-Trace to Terminating Program – Example

input: ultimately periodic trace

```
i>0
j:=1
( j<i
  j++)
```

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

```
i>0
j:=1
( j<i
  j++)
```
From $\omega$-Trace to Terminating Program – Example

input: ultimately periodic trace

1. consider $\omega$-trace as program with single while loop

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

3. compute rank certificate

4. add additional transitions

Implemented in Ultimate Büchi 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

- verification tasks
- optimized inclusion check for Büchi automata
- different ω-automata in termination analysis