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

Block		
Introduction	L 1:	20.4., Mo
	T 1:	23.4., Do
Development Process, Metrics	L 2:	27.4., Mo
	L 3:	30.4., Do
	L 4:	4.5., Mo
	T 2:	7.5., Do
	L 5:	11.5., Mo
Requirements Engineering	-	14.5., Do
	L 6:	18.5., Mo
	L 7:	21.5., Do
	-	25.5., Mo
	-	28.5., Do
	T 3:	1.6., Mo
	-	4.6., Do
	L 8:	8.6., Mo
	L 9:	11.6., Do
	L 10:	15.6., Mo
	T 4:	18.6., Do
Architecture & Design, Software Modelling	L 11:	22.6., Mo
	L 12:	25.6., Do
	L 13:	29.6., Mo
	L 14:	2.7., Do
	T 5:	6.7., Mo
Quality Assurance	L 15:	9.7., Do
	L 16:	13.7., Mo
Invited Talks	L 17:	16.7., Do
	T 6:	20.7., Mo
Wrap-Up	L 18:	23.7., Do

The Wireless Fire Alarm System:
Ensuring Conformance to Industrial Standards
through Formal Verification

Sergio Feo-Arenis Bernd Westphal Daniel Dietsch
Marco Muñiz Siyar Andisha

Software Engineering
Albert-Ludwigs-University Freiburg

July 16th – 2015

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?

Scenario

Develop a Standard-compliant Fire Alarm System
- Use a wireless protocol that supports range extenders (repeaters).
- Maximize energy efficiency.
- Ensure compliance with the norm DIN EN-54 (Part 25).

Scenario

EN-54 Requirements

- Detect and display communication failures in at most 300+100 seconds.
- Display alarms timely:
  - In at most 10 seconds for single alarms.
  - The first in 10 seconds and the last in 100 seconds for 10 simultaneous.
- Fulfill even when there are other users of the frequency.

Scenario

EN-54 Requirements

- Detect and display communication failures in at most 300+100 seconds.
- Display alarms timely:
  - In at most 10 seconds for single alarms.
  - The first in 10 seconds and the last in 100 seconds for 10 simultaneous.
- Fulfill even when there are other users of the frequency.

Scenario

EN-54 Requirements

- Detect and display communication failures in at most 300+100 seconds.
- Display alarms timely:
  - In at most 10 seconds for single alarms.
  - The first in 10 seconds and the last in 100 seconds for 10 simultaneous.
- Fulfill even when there are other users of the frequency.

Scenario

EN-54 Requirements

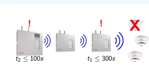
- Detect and display communication failures in at most 300+100 seconds.
- Display alarms timely:
  - In at most 10 seconds for single alarms.
  - The first in 10 seconds and the last in 100 seconds for 10 simultaneous.
- Fulfill even when there are other users of the frequency.

Scenario

EN-54 Requirements

- Detect and display communication failures in at most 300+100 seconds.
- Display alarms timely:
  - In at most 10 seconds for single alarms.
  - The first in 10 seconds and the last in 100 seconds for 10 simultaneous.
- Fulfill even when there are other users of the frequency.

Scenario



EN-54 Requirements

- Detect and display communication failures in at most 300+100 seconds.
- Display alarms timely:
  - In at most 10 seconds for single alarms.
  - The first in 10 seconds and the last in 100 seconds for 10 simultaneous.
- Fulfill even when there are other users of 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.

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.

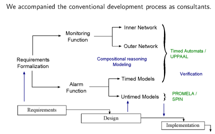
Thus:
Verification could help.

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.

Overview

We accompanied the conventional development process as consultants.



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

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 certificate authority.
- Required developing a graphical means of communication between the stakeholders. [Visual Narratives]

What to Verify: Requirements Formalization

$$\begin{aligned}\exists\, t_0, t_1, t_2, t_3 \bullet t_0 \le t_1 \le t_2 \le t_3 \\ \wedge FS(t_0) \wedge Disab_S(t_1) \wedge Det_S(t_2) \wedge Disp_S(t_3) \\ \wedge t_2 \le t_1 + 300 \wedge t_3 \le t_2 + 100 \\ \wedge \forall t \bullet t \ge t_0 \wedge t \ne t_1 \wedge t \ne t_2 \wedge t \ne t_3 \implies \emptyset(t)\end{aligned}$$

$$\downarrow$$

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.

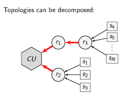
What to Verify: Requirements Formalization

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

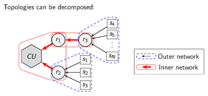
Modeling: Monitoring Function

Topologies can be decomposed:



Modeling: Monitoring Function

Topologies can be decomposed:



We modeled each "network" separately using networks of timed automata (UPPAAL).

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:

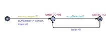
- No interference between networks (by design).
- No collisions (TDMA). [Guard time analysis]
- Topology subsumption: Verifying a maximal subnetwork is enough.

To make models tractable, we require optimization:

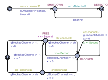
- Each component has an individual clock. [Quasi-equal clock reduction]
- Support plug-in models: Separate environment and design.

Modeling: Sensor Failures

Modeled as timed automata networks with UPPAAL:



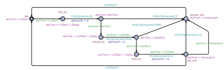
× 1



× 1

Modeling: Sensor Failures

× 4



× 126

Verification: Monitoring Function

Other model components:

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

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

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:

Query	Sensors as slaves			Repeaters as slaves		
	seconds	**MB**	**States**	**seconds**	**MB**	**States**
Detection	36,070.78	3,419.00	190M	231.84	230.59	6M
No Spurious	97.44	44.29	0.6M	3.94	10.14	0.15M
No LZ-Collision	12,895.17	2,343.00	68M	368.58	250.91	9.6M
Detection Possible	10,205.13	557.00	26M	38.21	55.67	1.2M

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

Verification: Alarm Function

For single, explicit topologies: Timed automata / UPPAAL.

Full collision				
Query	**ids**	**seconds**	**MB**	**States**
OneAlarm	-	3.6 ± 1	43.1 ± 1	$59k \pm 15k$
TwoAlarms	seq	4.7	67.1	110,207
TenAlarms	seq	44.6 ± 11	311.4 ± 102	$641k \pm 159k$
	opt	41.8 ± 10	306.6 ± 80	$600k \pm 140k$

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]

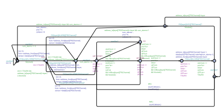
Models are still useful for simulation: extracted expected alarm times for different scenarios.

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.

Modeling: Alarm Function

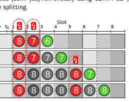


Verification: Alarm Function

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

Modeling: Alarm Function

Alarms are transmitted (semi-)asynchronously using CSMA-CD / Collision resolution using tree splitting.



Each component ID induces a unique timing pattern for retrying transmissions.

Verification: Alarm Function

For single, explicit topologies: Timed automata / UPPAAL.

Limited Collision				
Query	**ids**	**seconds**	**MB**	**States**
OneAlarm	-	1.4 ± 1	38.3 ± 1	$36k \pm 14k$
TwoAlarms	seq	0.5	24.1	19,528
TenAlarms	seq	17.3 ± 6	179.1 ± 61	$419k \pm 124k$
	opt	17.1 ± 6	182.2 ± 64	$412k \pm 124k$

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.

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

$\|I\|$	N	sec.	MB	States
255	2	49	1,610	1,235,970
H	10	3,393	6,390	6,242,610
L	10	4,271	10,685	10,439,545
Rnd	10	4,465	11,534	11,268,368
average		4,138	9,994	9,763,809

Lessons Learned

Generalized test procedures are useful for verification:

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

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 accesible for developers.

Formal verification shows potential to relieve the effort of testing.

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.

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).

salomo

Towards Successful Su contracting for Software in S all to ediu -Si ed Enter rises

REL orksho 2012-0 -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 Meierhöfer[3]

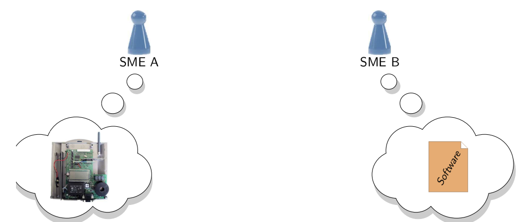
[1] Albert-Ludwigs-Universität Freiburg, Germany
[2] Universität des Saarlandes, Saarbrücken, Germany
[3] Universität Mannheim, Germany

Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg — UNIVERSITÄT MANNHEIM — ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

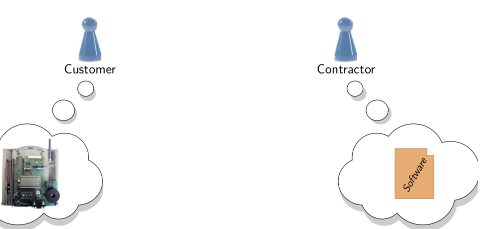
utline

- 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

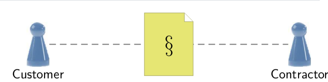
Successful Su contracting for Software in S Es



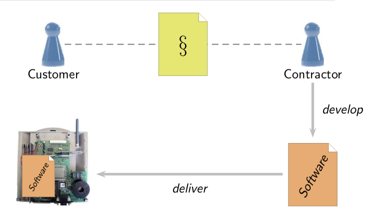
Successful Su contracting for Software in S Es



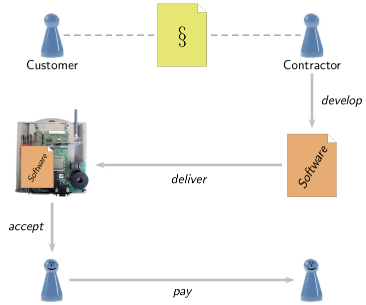
Successful Su contracting for Software in S Es



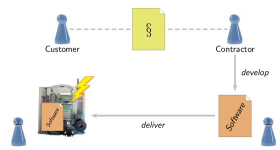
Successful Su contracting for Software in S Es



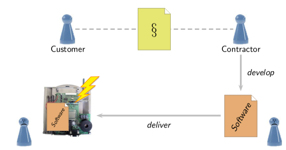
Successful Su contracting for Software in S Es



Su contracting for Software in S Es in Realit



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 is utes to ourt

... is generally **highly unattractive** for SME:

Bringing Software-related is utes to ourt

... is generally **highly unattractive** for SME:

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

Bringing Software-related is utes to ourt

... 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 is utes to ourt

. . . 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,

Bringing Software-related is utes to ourt

. . . 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 is utes to ourt

. . . 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**.

Bringing Software-related is utes to ourt

. . . 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,

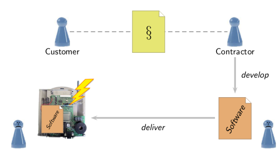
(iv) a court **only** decides over the **rights and duties** of each party,
no suggestion how to use the decision to achieve **project success**,

(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**.

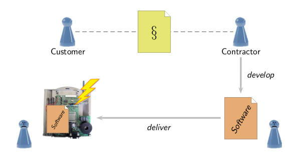
Su contracting for Software in S Es in Realit



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

- **misunderstandings** in the **requirements**,

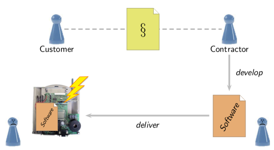
Su contracting for Software in S Es in Realit



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

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

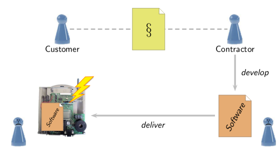
Su contracting for Software in S Es in Realit



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**.

Su contracting for Software in S Es in Realit



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**.

servation

- **(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.

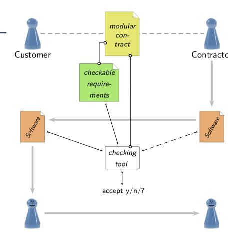
(c) **standardised legal contracts** which integrate (a) and (b),
they avoid the 3rd source of uncertainty.

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

utline

- 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

Towards Legal ertaint



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**.

hecka le S eci cation/Re uire ent hecking Tool

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

- A **backend** maps a program p and a program property φ
  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.

hecka le S eci cation/Re uire ent hecking Tool

- A **checkable specification** is a pair (φ, T) comprising a **program property** φ and a **backend** T.
- A **backend** maps a program p and a program property φ 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 **checking tool** maps a set of checkable specifications

$$\Phi = \{(\varphi_1, T_1), \dots, (\varphi_n, T_n)\}, n \in \mathbb{N}_0,$$

to a **checking tool result**

$$\{(\varphi_1, s_1), \dots, (\varphi_n, s_n)\}, s_i \in \{Yes, No, Unknown\}.$$

hecka le S eci cation/Re uire ent hecking Tool

- A **checkable specification** is a pair (φ, T) comprising a **program property** φ and a **backend** T.
- A **backend** maps a program p and a program property φ 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 **checking tool** maps a set of checkable specifications

$$\Phi = \{(\varphi_1, T_1), \dots, (\varphi_n, T_n)\}, n \in \mathbb{N}_0,$$

to a **checking tool result**

$$\{(\varphi_1, s_1), \dots, (\varphi_n, s_n)\}, s_i \in \{Yes, No, Unknown\}.$$

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

Backend E a les

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

Backend E a les

- **"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 E a les

- **"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.

Backend E a les

- **"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 E a les

- **"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 ontract

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

Regulations in the ontract

- 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.

utline

- 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 ork

- (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.

onclusion and urther ork

- 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.

Software Verification

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

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

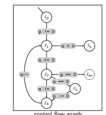
Thanks.

http://www.salomo-projekt.de

Example

```
ℓ0:  assume p != 0;
ℓ1:  while(n >= 0)
     {
ℓ2:      assert p != 0;
         if(n == 0)
         {
ℓ3:          p := 0;
         }
ℓ4:      n--;
     }
```

pseudocode



control flow graph

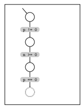
ULTIMATE

Automizer

Software Verification

- prove or disprove that a given program satisfies a given specification
- problem is undecidable [Turing, 1936]

1. take trace π_1
2. consider trace as program $\mathcal{P}_1$

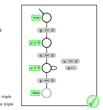



```
1:  assume p != 0;
2:  assume n >= 0;
3:  assert p != 0;
```

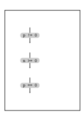
pseudocode of $\mathcal{P}_1$

1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions

{ $p \neq 0$ } `n--` { $p \neq 0$ } is valid Hoare triple
{ $p \neq 0$ } `n != 0` { $p \neq 0$ } is valid Hoare triple

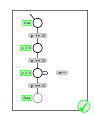


1. take trace π_1



1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions

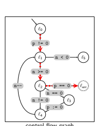
{ $p \neq 0$ } `n--` { $p \neq 0$ } is valid Hoare triple



Example

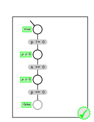
```
ℓ0:  assume p != 0;

ℓ1:  while(n >= 0)
     {
ℓ2:      assert p != 0;

         if(n == 0)
         {
ℓ3:          p := 0;
         }
ℓ4:      n--;
     }
```

pseudocode

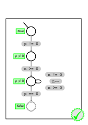


control flow graph

1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$

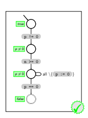


1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions

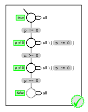


$\{p \neq 0\}$ `n--` $\{p \neq 0\}$ is valid Hoare triple
$\{p \neq 0\}$ `n != 0` $\{p \neq 0\}$ is valid Hoare triple
$\{p \neq 0\}$ `n >= 0` $\{p \neq 0\}$ is valid Hoare triple

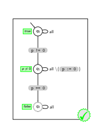
1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions

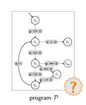


1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions

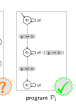


1. take trace π_1
2. consider trace as program $\mathcal{P}_1$
3. analyze correctness or $\mathcal{P}_1$
4. generalize program $\mathcal{P}_1$
   - add transitions
   - merge locations





program $\mathcal{P}$



program $\mathcal{P}_1$

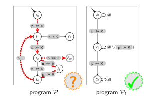
New View on Programs

"A program defines a language over the alphabet of statements."

New View on Programs

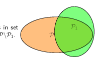
“A program defines a language over the alphabet of statements.”

- Set of statements: alphabet of formal language
  e.g., $\Sigma = \{$ `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
- Error trace of program: word accepted by this automaton



program $\mathcal{P}$ program $\mathcal{P}_1$

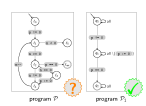
Consider only traces in set theoretic difference $\mathcal{P} \setminus \mathcal{P}_1$



New View on Programs

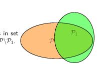
“A program defines a language over the alphabet of statements.”

- Set of statements: alphabet of formal language
  e.g., $\Sigma = \{$ `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



program $\mathcal{P}$ program $\mathcal{P}_1$

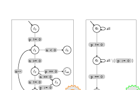
Consider only traces in set theoretic difference $\mathcal{P} \setminus \mathcal{P}_1$



New View on Programs

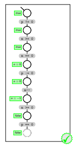
“A program defines a language over the alphabet of statements.”

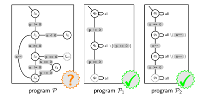
- Set of statements: alphabet of formal language
  e.g., $\Sigma = \{$ `p != 0`, `n >= 0`, `n == 0`, `p := 0`, `n != 0`, `p == 0`, `n--`, `n < 0`, $\}$



program $\mathcal{P}$ program $\mathcal{P}_1$

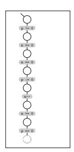
1. take trace π_2
2. consider trace as program $\mathcal{P}_2$
3. analyze correctness or $\mathcal{P}_2$

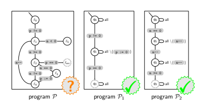




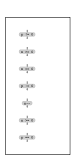
$$\mathcal{P} \quad \subseteq \quad \mathcal{P}_1 \cup \mathcal{P}_2$$

1. take trace π_2
2. consider trace as program $\mathcal{P}_2$

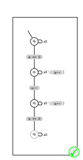




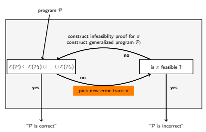
1. take trace π_2



1. take trace π_2
2. consider trace as program $\mathcal{P}_2$
3. analyze correctness or $\mathcal{P}_2$
4. generalize program $\mathcal{P}_2$
   - add transitions
   - merge locations

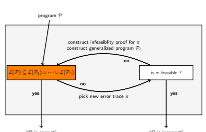


Verification Algorithm

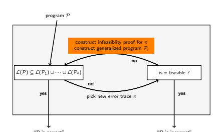


Interprocedural/Recursive Programs

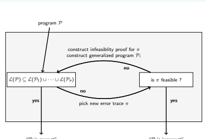
Verification Algorithm



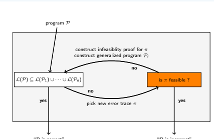
Verification Algorithm



Verification Algorithm



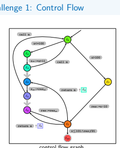
Verification Algorithm



Recursive Programs - Challenge 1: Control Flow

```
procedure m(x) returns (res)
ℓ0: if x>100
ℓ1:   res:=x-10
    else
ℓ2:   xm := x+11
ℓ3:   call m
ℓ4:   xm := resm
ℓ5:   call m
ℓ6:   res := resm
ℓ7: assert (x<=101 -> res=91)
    return m
```

McCarthy 91 function

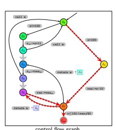


control flow graph

Recursive Programs - Challenge 1: Control Flow

```
procedure m(x) returns (res)
ℓ0: if x>100
ℓ1:   res:=x-10
    else
ℓ2:   xm := x+11
ℓ3:   call m
ℓ4:   xm := resm
ℓ5:   call m
ℓ6:   res := resm
ℓ7: assert (x<=101 -> res=91)
    return m
```

McCarthy 91 function



control flow graph

Recursive Programs - Challange 2: Local Annotations

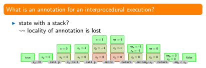
What is an annotation for an interprocedural execution?

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

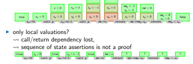
Recursive Programs - Challange 2: Local Annotations

What is an annotation for an interprocedural execution?

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



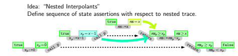
- only local valuations?
  ⇝ call/return dependency lost,
  ⇝ sequence of state assertions is not a proof



Recursive Programs - Challange 2: Local Annotations

What is an annotation for an interprocedural execution?

Idea: “Nested Interpolants”
Define sequence of state assertions with respect to nested trace.



Define ternary post operator for return statements

$$post(\boxed{res = x}, \boxed{x_p = x - 1}, \texttt{return p}) \subseteq \boxed{res_p \geq x_p}$$

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 ω-words and Büchi automata

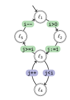
- Challenge 2: An infinite trace may not have any execution although each finite prefix has an execution.

  E.g., $($ `x > 0` `x--` $)^\omega$

```
while (x > 0) {
    x--;
}
```

Example: Bubble Sort

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



Termination Analysis

- Challenge 1: counterexample to termination is infinite execution

  Solution: consider infinite traces, use ω-words and Büchi automata

Example: Bubble Sort

```
program sort(int i, int a[])
ℓ1 while (i>0)
ℓ2     int j:=1
ℓ3     while(j<i)
           if (a[j]>a[i])
               swap(a,i,j)
ℓ4         j++
ℓ5     i--
```

Termination Analysis

- Challenge 1: counterexample to termination is infinite execution

Termination Analysis

- Challenge 1: counterexample to termination is infinite execution

  Solution: consider infinite traces, use ω-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--` $)^\omega$

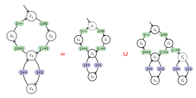
```
while (x > 0) {
    x--;
}
```

  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.

program $\mathcal{P}$		module $\mathcal{P}_1$		module $\mathcal{P}_2$
$(\textsc{Outer}+\textsc{Inner})^\omega$	=	$(\textsc{Inner}^*.\textsc{Outer})^\omega$	+	$(\textsc{Inner}+\textsc{Outer})^*.\textsc{Inner}^\omega$



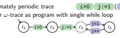
ranking function
$f(\mathtt{i},\mathtt{j}) = \mathtt{i}$

ranking function
$f(\mathtt{i},\mathtt{j}) = \mathtt{i} - \mathtt{j}$

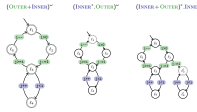
From ω-Trace to Terminating Program – Example

input: ultimately periodic trace `i>0` `j:=1` (`j<i` `j++`)$^\omega$,

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



program $\mathcal{P}$	module $\mathcal{P}_1$	module $\mathcal{P}_2$
$(\textsc{Outer}+\textsc{Inner})^\omega$	$(\textsc{Inner}^*.\textsc{Outer})^\omega$	$(\textsc{Inner}+\textsc{Outer})^*.\textsc{Inner}^\omega$



ranking function
$f(\mathtt{i},\mathtt{j}) = \mathtt{i}$

ranking function
$f(\mathtt{i},\mathtt{j}) = \mathtt{i} - \mathtt{j}$

From ω-Trace to Terminating Program – Example

input: ultimately periodic trace `i>0` `j:=1` (`j<i` `j++`)$^\omega$,

Example: Bubble Sort

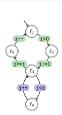
```
program sort(int i)
ℓ1 while (i>0)
ℓ2     int j:=1
ℓ3     while(j<i)
ℓ4         j++
ℓ5     i--
```

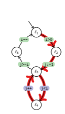
quadratic ranking function:

$$f(\mathtt{i},\mathtt{j}) = \mathtt{i}^2 - \mathtt{j}$$

lexicographic ranking function:

$$f(\mathtt{i},\mathtt{j}) = (\mathtt{i}, \mathtt{i} - \mathtt{j})$$

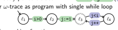




From ω-Trace to Terminating Program – Example

input: ultimately periodic trace `i>0` `j:=1` (`j<i` `j++`)$^\omega$,

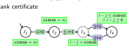
1. consider ω-trace as program with single while loop



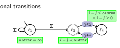
2. synthesize ranking function

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

3. compute rank certificate



4. add additional transitions



Implemented in

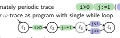
Ultimate Büchi Automizer

`http://ultimate.informatik.uni-freiburg.de/BuchiAutomizer/`

From ω-Trace to Terminating Program – Example

input: ultimately periodic trace `i>0` `j:=1` (`j<i` `j++`)$^\omega$,

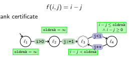
1. consider ω-trace as program with single while loop



2. synthesize ranking function

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

3. compute rank certificate



Generalization of Program with Rank Certificate

- Case 1: q_1 not accepting

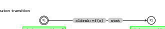
  Hoare triple { `state assertion 1` } `stmt` { `state assertion 2` }

  automaton transition

- Case 2: q_1 accepting

  Hoare triple { `state assertion 1` } `oldrnk:=f(x)` `stmt` { `state assertion 2` }

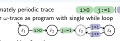
  automaton transition



From ω-Trace to Terminating Program – Example

input: ultimately periodic trace `i>0` `j:=1` (`j<i` `j++`)$^\omega$,

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



2. synthesize ranking function

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

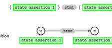
Colón, Sipma **Synthesis of Linear Ranking Functions** (TACAS 2001)
Podelski, Rybalchenko **A complete method for the synthesis of linear ranking functions** (VMCAI 2004)
Bradley, Manna, Sipma **Termination Analysis of Integer Linear Loops** (CONCUR 2005)
Bradley, Manna, Sipma **Linear ranking with reachability** (CAV 2005)
Bradley, Manna, Sipma **The polyranking principle** (ICALP 2005)
Ben-Amram, Genaim **Ranking functions for linear-constraint loops** (POPL 2013)
H., Hoenicke, Leike, Podelski **Linear Ranking for Linear Lasso Programs** (ATVA 2013)
Cook, Kroening, Rümmer, Wintersteiger **Ranking function synthesis for bit-vector relations** (FMSD 2013)
Leike, H. **Ranking Templates for Linear Loops** (TACAS 2014)

Generalization of Program with Rank Certificate

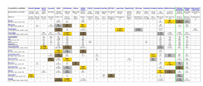
- Case 1: q_1 not accepting

  Hoare triple { `state assertion 1` } `stmt` { `state assertion 2` }

  automaton transition



Results of the Competition on Software Verification 2015



Thank you for your attention!

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

Implemented in

Ultimate Büchi Automizer

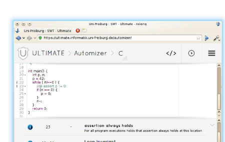
`http://ultimate.informatik.uni-freiburg.de/BuchiAutomizer/`

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



`http://ultimate.informatik.uni-freiburg.de/automizer`