

VL15	• Introduction and Vocabulary
VL16	• Limits of Software Testing
VL17	• Glass-Box Testing
VL18	• Statement, branch, item-coverage
VL19	• Other Approaches
VL20	• Model-based testing
VL21	• Runtime verification
VL22	• Software quality assurance
VL23	• n a target scope
VL24	• Program Verification
VL25	• partial analytical correctness,
VL26	• proof system PQ
VL27	• Runtime Verification
VL28	• Review
VL29	• Code QA Discussion

• Runtime Verification
• Ideas
• Assertions
• LSC-Observers
• Reviews
• Roles and artefacts
• Review procedure
• Stronger and weaker variants
• Do's and Don'ts in Code QA
• Code QA techniques Revisited
• Test
• Runtime Verification
• Review
• State Checking
• Formal Verification
• Dependability

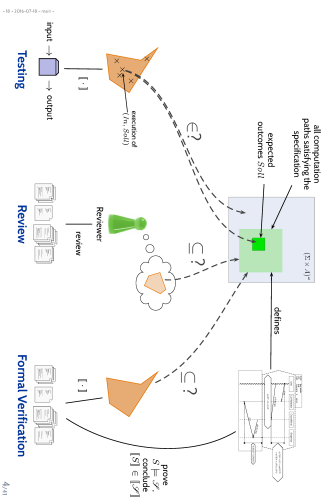
Software-Engineering Lecture 18: Runtime Verification, Review & Wrapup

2016-07-18

Prof. Dr. Andreas Poddicki, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Recall: Three Basic Directions



Run-Time Verification

Run-Time Verification: Idea

- Assume there is a function f in software S with the following specification:
 - pre-condition p , post-condition q .
 - Computation paths of S may look like this:
- $$\sigma_0 \xrightarrow{\text{call } f} \sigma_1 \xrightarrow{\text{call } f} \sigma_2 \dots \xrightarrow{\text{call } f} \sigma_{m-1} \xrightarrow{\text{call } f} \sigma_m \xrightarrow{\text{call } f} \sigma_{m+1} \dots$$
- Assume there are functions $check_p$ and $check_q$, which check whether p and q hold at the current program state, and which **do not modify the program state** (except for program counter).
 - Idea: create software S' by
 - extending S by implementations of $check_p$ and $check_q$.
 - call $check_p$ right after entering f .
 - call $check_q$ right before returning from f .
 - For S' , obtain computation paths like:
- $$\sigma_0 \xrightarrow{\text{call } f} \sigma_1 \xrightarrow{\text{call } f} \sigma_2 \dots \xrightarrow{\text{call } f} \sigma_{m-1} \xrightarrow{\text{call } f} \sigma_m \xrightarrow{\text{call } f} \sigma_{m+1} \dots$$
- If $check_p$ and $check_q$ notify us of violations of p or q , then we are notified of **violating its specification when running S'** (= at run-time).

Run-Time Verification: Example



```
int main() {
    while (true) {
        int x = rand_number();
        int y = rand_number();
        int sum = add(x, y);
        verify_sum(x, y, sum);
        display(sum);
    }
}
```

```
void verify_sum(int x, int y, int sum) {
    if (sum != x+y) {
        fprintf(stderr, "Error: sum is %d, expected %d\n", sum, x+y);
        abort();
    }
}
```

7/8

A Very Useful Special Case: Assertions

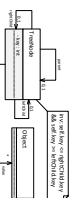
- Maybe the simplest instance of runtime verification: **Assertions**.
- Available in standard libraries of many programming languages (C, C++, Java, ...).
- For example, the C standard library manual reads:

```
ASSERTIONS
Name: assert
Synopsis: assert (assert)
Description: The assert macro is used to assert that a condition is true. If the condition is false, the program will abort. The purpose of the macro is to help the programmer find errors in his code. Do not use assert in production code.
```

- In C code asserts can be **disabled in production code** (-D NDEBUG).

8/8

Assertions At Work II



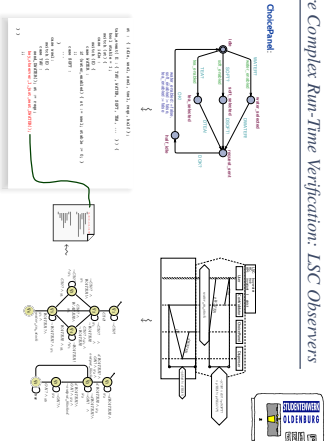
- Recall the **structure model** with Prop-OCL constraint from Exercise Sheet 4.
- Assume, we add a method `set_key()` to class **TreeNode**:

```
class TreeNode {
    ...
    public void set_key( int new_key ) {
        ...
    }
}
```
- We can check consistency with the Prop-OCL constraint at runtime by using assertions:

```
public void set_key( int new_key ) {
    assert ( parent != null || parent.get_key() != new_key );
    assert ( rightChild == null || new_key < rightChild.get_key() );
    key = new_key;
}
```
- Use Java -ea ... to enable assertion checking (disabled by default).
<https://docs.oracle.com/javase/8/docs/technotes/guides/annotations/assert.html>

10/11

More Complex Run-Time Verification: LSC Observers



11/11

Assertions At Work

- The abstract *f*-example from run-time verification
- Compute the width of a progress bar *index* *46*

```
void f(int i) {
    ...
    assert ( i < 100 );
}
```



```
int progress_bar_width( int progress, int width, int index ) {
    assert ( index <= width );
    assert ( progress <= width );
    assert ( index <= progress );
    assert ( width > 0 );
    return width - index;
}
```

9/11

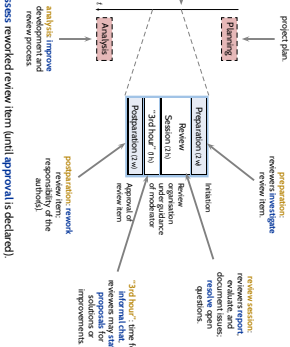
Run-Time Verification: Discussion

- **Experience:**
During development, assertions for pre/post conditions and intermediate invariants are an **extremely powerful** tool with a very attractive gain/effort ratio (low effort, high gain).
Assertions effectively work as **safety guard** against unexpected use of functions and **regression**.
E.g. during later maintenance or efficiency improvement.
Can serve as **formal** (support of) **documentation**.
Dear reader, at this point in the program, **expect condition expr to hold** because: ...
- **Development- vs. Release Versions**
- **Common practice**
 - development version **with** run-time verification enabled (cf. assert(3)).
 - release version **without** run-time verification.
- If run-time verification is enabled in a release version,
 - software should **terminate** as gracefully as possible e.g. try to save data.
 - **save information** from assertion failure if possible for future analysis.
- **Risk:** with bad luck, the software only behaves well **because** of the run-time verification code. Then disabling run-time verification "breaks" the software. Yet very complex run-time verification may significantly slow down the software, so needs to be disabled.

12/11

- Runtime-Verification
 - Idea
 - Assertions
 - LSC-Observers
- Reviews
 - Roles and artefacts
 - Review procedure
 - Stronger and weaker variants
- Do's and Don'ts in Code QA
- Code QA Techniques Revisited
 - Test
 - Runtime-Verification
 - Review
 - Static Checking
 - Formal Verification
- Dependability

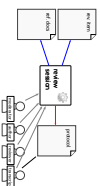
Review Procedure Over Time



Review

- (i) The moderator organises the review, issues invitations, supervises the review session.
- (ii) The moderator may terminate the review if conduct is not possible, e.g. due to interrupted preparation, or preparing missing.
- (iii) The review session is limited to 2 hours, if needed, organise more sessions.
- (iv) The review item is under review, not the author's.
- (v) Reviewers need to read needs accurately. Authors need to defend their views to the review item.
- (vi) Reviewers are **not** reviewing, e.g. the moderator does not act as reviewer (exception: author may write transcript)
- (vii) **Style** issues (outside fixed conventions) are **not** discussed.
- (viii) The review team is **not** supposed to discuss issues **not** noted down in form of tasks for the author(s).
- (ix) Each reviewer gets the opportunity to present his/her findings appropriately.
- (x) Reviewers need to reach consensus on issues, consensus is noted down.
- (xi) Issues are classified as:
 - critical (reviewer unusable for proposal)
 - major (quality severely affected)
 - minor (quality slightly affected)
 - good (no problem)
- (xii) The review team declares:
 - **accept with changes**;
 - **accept**;
 - **do not accept**.
- (xiii) The protocol is signed by all participants.

Reviews




- **Input to Review Session:**
 - Review item (e.g. software code, human-readable part for formal verification, module test data, installation manual, etc.)
 - **Social aspect:** It can be **artificial** (e.g. code review) or **real** (e.g. code review).
 - **Reference documents:** need to enable an assessment (e.g. coding conventions, catalogue of questions (cf. **analyzer**)).
- **Rules:**
 - **Moderator:** leads session, responsible for properly conducted procedure.
 - **Author:** (representative of the creator) of the artefact under review: is present to **clarify** the discussions.
 - **Reviewer:** (representative of the reviewer) of the artefact under review: is present to **clarify** the discussions.
 - **Reviewer:** (representative of the reviewer) of the artefact under review: is present to **clarify** the discussions.
 - **Reviewer:** (representative of the reviewer) of the artefact under review: is present to **clarify** the discussions.
- The review team consists of everybody but the author(s).


Stronger and Weaker Review Variants

- **Design and Code Inspection** (Fagan, 1976, 1981)
 - **Design:** 100% more time, approx. 100% more errors found.
 - **Code:** 100% more time, approx. 100% more errors found.
- **Review**
 - **Structured Walkthrough:**
 - single version of review
 - developer moderate walkthrough questions
 - reviewer poses prepared or spontaneous questions
 - developer presents artefact(s)
 - reviewer provides feedback
 - reviewer and developer are the artefact before the session?
 - **drawback:** underresponsibilities, 'talker'-developer may risk reviews
 - **Comment (Stellungnahme)**
 - colleagues of developer and artefacts
 - developer considers feedback
 - **drawback:** low organisational effort, consideration of comments at discretion of developer
 - **Careful Reading (Düchtheit)**
 - done by developer
 - recommendation: 'many from screen' (use print-out or different device and situation)


Do's and Don'ts in Code Quality Assurance



Avoid using special examination versions for examination. (Test-frames, stubs, etc. may have errors which may cause false positives and false negatives.)




Avoid to stop examination when the first error is detected. Clear: Examination should be aborted if the examined program is not executable at all.




Do not modify the artefact under examination **during** examination. otherwise, it is unclear what exactly has been examined ("moving target"). (examination results need to be uniquely traceable to one artefact version)

- fundamental flaws are sometimes easier to detect
- changes are particularly error-prone: should not happen "en passant" in examination.
- changes are particularly error-prone: should not happen "en passant" in examination. (with the need for all kinds of estimation)



Do not switch (freely gained) between examination and debugging.



Do not switch (freely gained) between examination and debugging.

Some Final, General Guidelines

Content

- Runtime-Verification
 - Ideas
 - Assertions
 - LSC-Observers
- Reviews
 - Roles and artefacts
 - Review procedure
 - Stronger and weaker variants
- Do's and Don'ts in Code QA
- Code QA techniques Revised
 - Test
 - Runtime-Verification
 - Review
 - Static Checking
 - Formal Verification
- Dependability

Techniques Revised

	auto- checked	prove "can show" checked	toolchain checked	exhaustive checked	prove checked	partial checked	entry checked
Test	(✓)	✓	✓	✗	✗	✓	✓
Runtime- Verification							
Review							
Static Checking							
Verification							

- Strengths:**
- can be fully automated (type not only for C/C++ programs)
 - can be used for "proving" the correctness of a program (e.g., "can run" for positive examples)
 - final product is examined: thus toolchain and platform considered
 - one can stop at any time and take partial results
 - few simple test cases are usually easy to obtain
 - provides reproducible counter-examples (good starting point for repair)
- Weaknesses:**
- (almost) cannot safely incomplete, thus no proof of correctness
 - creating test cases for complex functions for complex conditions can be difficult
 - examining many tests may need substantial time (but can sometimes be run in parallel)

Techniques Revised

	auto- checked	prove "can show" checked	toolchain checked	exhaustive checked	prove checked	partial checked	entry checked
Test	(✓)	✓	✓	✗	✗	✓	✓
Runtime- Verification							
Review							
Static Checking							
Verification							

- Strengths:**
- may be automated (focus on how to be placed)
 - can be used for "proving" the correctness of a program (e.g., "can run" for positive examples)
 - final product is examined: thus toolchain and platform considered
 - one can stop at any time and take partial results
 - few simple test cases are usually easy to obtain
 - provides reproducible counter-examples (good starting point for repair)
- Weaknesses:**
- counter-examples not necessarily reproducible
 - may negatively affect performance
 - code is changed: program may only run because of the observers
 - may also be safely incomplete as no correctness proofs
 - conducting observers for complex properties may be difficult
 - overhead to learn how to construct observers

Techniques Revisited

	also- possible	prove 'can conform'	toolchain coverage	exhaustive coverage	prove correctness	partial coverage	entry point
Test	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)
Runtime- Verification	(✓)	(✓)	(✓)	(X)	(X)	(✓)	(✓)
Review	(X)	(X)	(X)	(✓)	(✓)	(✓)	(✓)
Static Checking Verification	(X)	(X)	(X)	(✓)	(✓)	(✓)	(✓)

Strengths:

- human readers can understand the code, may spot/point errors;
- reported to be highly effective;
- can be automated, but not fully automated;
- intermediate entry costs;
- good effort/effect ratio achievable.

Weaknesses:

- no tool support;
- no results on actual execution, toolchain not reviewed;
- no results on actual execution, tools usually not arriving at proofs;
- does (by general) not provide coverage analysis;
- developers may deny existence of error.

23.0

Techniques Revisited

	also- possible	prove 'can conform'	toolchain coverage	exhaustive coverage	prove correctness	partial coverage	entry point
Test	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)
Runtime- Verification	(✓)	(✓)	(✓)	(X)	(X)	(✓)	(✓)
Review	(X)	(X)	(X)	(✓)	(✓)	(✓)	(✓)
Static Checking Verification	(✓)	(X)	(X)	(✓)	(✓)	(✓)	(X)

Strengths:

- these are (commercial, fully automatic tools like Coverity, Polyspace, etc.);
- some tools are complete (relative to assumptions on language semantics, platform, etc.);
- can be automated, but not fully automated;
- one can stop at any time and take partial results.

Weaknesses:

- no results on actual execution, toolchain not reviewed;
- can be very resource consuming (if few false positives wanted);
- E.g. coding may need to be designed for static analysis;
- entry cost high, significant learning is useful to know how to deal with tool limitations;
- defining false from true conditions can be challenging;
- configuring the tools (to find false positives) can be challenging.

23.1

Techniques Revisited

	also- possible	prove 'can conform'	toolchain coverage	exhaustive coverage	prove correctness	partial coverage	entry point
Test	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)
Runtime- Verification	(✓)	(✓)	(✓)	(X)	(X)	(✓)	(✓)
Review	(X)	(X)	(X)	(✓)	(✓)	(✓)	(✓)
Static Checking Verification	(✓)	(X)	(X)	(✓)	(✓)	(✓)	(X)

Strengths:

- some tools support toolable (fine-grained) tools;
- some tools belong to assumptions on language semantics, platform, etc.);
- that can provide correctness proofs;
- can prove correctness for multiple language semantics and platforms at a time;
- can be more efficient than other techniques.

Weaknesses:

- no results on actual execution, toolchain not reviewed;
- not many internalised are results, 'half of a proof' may not allow any useful conclusions;
- entry cost high, significant learning is useful to know how to deal with tool limitations;
- defining false from true conditions can be challenging;
- false positives: harder progress on 'proof' correct hard to detect.

23.2

Quality Assurance — Concluding Discussion

Proposed: Dependability Cases (Linkson, 2009)

- A dependable system is one you can depend on – that is, you can place your trust in it.

"Developers (should) express the critical properties and make an explicit argument that the system satisfies them."

quality assurance – (1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to set requirements.

Proposed Approach:

- Identify the critical requirements, and determine what level of confidence is needed.
- Most systems do also have non-critical requirements.
- Construct a dependability case.
- an argument, that the software, in concert with other components, establishes the critical properties.
- The case should be:
- auditable, can (easily) be evaluated by third-party certifier;
- complete, no holes in the argument, any assumptions that are not justified, should be modelled (e.g. assumptions on complex, on particular deployed by users, etc.);
- based on the evidence, evidence is independent of component failures etc.

24.1

Critical Systems

- Still, it seems like computer systems more or less inevitably have errors.

Then why...

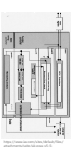


- ... do modern planes fly at all?
 - very careful development;
 - very thorough analysis;
 - very strict safety requirements.
 Plus, classic engineering wisdom for high reliability: live redundancy.



Figure 4.1.2009

- ... do modern cars drive at all?
 - careful development;
 - thorough analysis;
 - regulatory obligations.
 Plus, classic engineering wisdom for high reliability: live redundancy.



24.2

Tell Them What You've Told Them...

- **Runtime Verification**
 - **active** name suggests checks properties at **program run-time**.
 - a good **principle of aspects** can be a valuable safety guard against **errors**.
 - **assertions**.
 - **usage outside specification**.
 - **code**.
 - and serve as **formal documentation** of assumptions.
- **Review** (structured examination of artefacts by humans)
 - **find variant** associated in the XP approach.
 - **not uncommon**.
 - **lead programmer** reviews. All commits from team members.
 - **iterative** reports good effort/correct ratio achievable.
- **All approaches to code quality assurance** have their
 - **advantages** and **drawbacks**.
 - **Which to use?** It depends!
- **Dependency Cases**
 - an auditable, complete sound argument
 - that a software has the critical properties.

References

Figari, M. (1976). Design and code inspections to reduce errors in program development. IBM Systems Journal, 15(3), 182-211.

Figari, M. (1986). Advances in software inspections. IEEE Transactions On Software Engineering, 12(7), 744-751.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology. Std 610.12-1990.

Jackson, D. (2009). A direct path to dependable software. Comm. ACM, 52(4).

Ludewig, J. and Löhner, H. (2013). Software Engineering. 3rd edition.

Margala, C., Robbes, R., Schaefer, L., Toben, T. and Weispahl, B. (2009). Formal verification of a sensor networking and monitoring system. In Proceedings of the 2009 ACM/IEEE International Conference on Formal Engineering and Design (FED'09), pages 37-51. Technische Universität München.

Contents of the Course



Expectations

- more because mandatory course
- overall
 - well-structured
 - well-thought-out
 - clearly defined objectives
 - clearly defined learning objectives
 - emphasized scientific work
 - emphasized scientific methods
 - other courses
 - research how courses are linked together
 - emphasized knowledge in SDs
 - emphasized knowledge in SDs
- very useful
 - inculcated and embedded in professional software development
 - learn how things work in a company to make things into a team & communication
 - team & communication
- blend of software
 - embedded systems and software
 - how to combine HW and SW

Initiation	1	18.4	Mon
Scope Defn.	2	21.7	Tue
	3	24.4	Mon
	4	25.3	Mon
Development	5	25.3	Mon
Process	6	25.3	Mon
	7	25.3	Mon
	8	25.3	Mon
Requirements Engineering	9	25.3	Mon
	10	25.3	Mon
	11	25.3	Mon
Architecture & Design	12	25.3	Mon
	13	25.3	Mon
	14	25.3	Mon
Software Modeling	15	25.3	Mon
	16	25.3	Mon
	17	25.3	Mon
Quality Assurance	18	25.3	Mon
Testing Round	19	25.3	Mon
Wrap-Up	20	25.3	Mon

Aver

33/41

Expectations Cont'd

- Software development
 - understand the requirements and develop specific tasks
 - design, implement and test
 - deploy and maintain
- Software development process
 - define project management
 - How to create an organization of a project
 - How to keep control of a project, maintain a team
 - which kind of documentation is necessary
 - what is getting in the way? how to deal with it
 - what is getting in the way? how to deal with it
 - How to estimate the time and cost
 - How to estimate the time and cost
 - How to estimate the time and cost
- Quality
 - How to make a quality plan, based on the requirements
 - make a quality plan, based on the requirements
 - make a quality plan, based on the requirements

Introduction	1	1	18.4
Scale, Metrics, Costs	2	1	21.4
Development	3	1	28.4
Process	4	2	25.4
	5	1	35.4
	6	1	42.4
	7	1	49.4
	8	1	56.4
	9	1	63.4
	10	1	70.4
	11	1	77.4
	12	1	84.4
	13	1	91.4
	14	1	98.4
	15	1	105.4
	16	1	112.4
	17	1	119.4
	18	1	126.4
	19	1	133.4
	20	1	140.4
	21	1	147.4
	22	1	154.4
	23	1	161.4
	24	1	168.4
	25	1	175.4
	26	1	182.4
	27	1	189.4
	28	1	196.4
	29	1	203.4
	30	1	210.4
	31	1	217.4
	32	1	224.4
	33	1	231.4
	34	1	238.4
	35	1	245.4
	36	1	252.4
	37	1	259.4
	38	1	266.4
	39	1	273.4
	40	1	280.4
	41	1	287.4
	42	1	294.4
	43	1	301.4
	44	1	308.4
	45	1	315.4
	46	1	322.4
	47	1	329.4
	48	1	336.4
	49	1	343.4
	50	1	350.4
	51	1	357.4
	52	1	364.4
	53	1	371.4
	54	1	378.4
	55	1	385.4
	56	1	392.4
	57	1	399.4
	58	1	406.4
	59	1	413.4
	60	1	420.4
	61	1	427.4
	62	1	434.4
	63	1	441.4
	64	1	448.4
	65	1	455.4
	66	1	462.4
	67	1	469.4
	68	1	476.4
	69	1	483.4
	70	1	490.4
	71	1	497.4
	72	1	504.4
	73	1	511.4
	74	1	518.4
	75	1	525.4
	76	1	532.4
	77	1	539.4
	78	1	546.4
	79	1	553.4
	80	1	560.4
	81	1	567.4
	82	1	574.4
	83	1	581.4
	84	1	588.4
	85	1	595.4
	86	1	602.4
	87	1	609.4
	88	1	616.4
	89	1	623.4
	90	1	630.4
	91	1	637.4
	92	1	644.4
	93	1	651.4
	94	1	658.4
	95	1	665.4
	96	1	672.4
	97	1	679.4
	98	1	686.4
	99	1	693.4
	100	1	700.4
	101	1	707.4
	102	1	714.4
	103	1	721.4
	104	1	728.4
	105	1	735.4
	106	1	742.4
	107	1	749.4
	108	1	7

Se

34/4

Expectations Com'd

- [illegible]

Production	L1	184	
Scale, Perf., Cost	L2	173	
Code	L3	284	
Development	L4	231	
	L5	55	
Process	L5	916	
	L6	123	
		85	
		72	
		24	
Requirements Engineering	L2	303	
	L6	26	
Architectural Design	L3	96	
	L7	16	
	L8	60	
Software	L4	216	
Modeling	L5	30	
	L6	47	
	L7	72	
Quality Assurance	L1	112	
Testing Frameworks	L2	84	
Workflow	L3	217	
Wrap-Up	L8	27	

3

35/

Expectations Cont'd

- **code quality assurance**
 - methods for ensuring high level of quality
 - how to conduct most exhaustive test as possible in reasonable time
 - formal methodical program verification
 - learn about practical implementation of these tools
- **extra information**
 - "will work as teacher"
 - "want to work on medical software"
 - "want to work in automotive industry"
 - "worked as software engineer"

Installation	1	184	Mon
Scale Physica	2	201	Mon
Scale Physica	3	204	Mon
Scale Physica	4	204	Mon
Development	5	255	Mon
Process	6	95	Mon
	7	115	Mon
	8	85	Mon
	9	85	Mon
Requirements Engineering	10	305	Mon
	11	305	Mon
	12	26	Mon
	13	66	Mon
	14	96	Mon
Architecture Design	15	16	Mon
	16	26	Mon
	17	86	Mon
	18	206	Mon
	19	216	Mon
	20	216	Mon
Software Modeling	21	306	Mon
	22	306	Mon
	23	47	Mon
	24	27	Mon
Quality Assurance (Testing Loop)	25	187	Mon
	26	147	Mon
	27	147	Mon
	28	7	Mon
Wrap-Up	29	217	Mon

7/10

3641

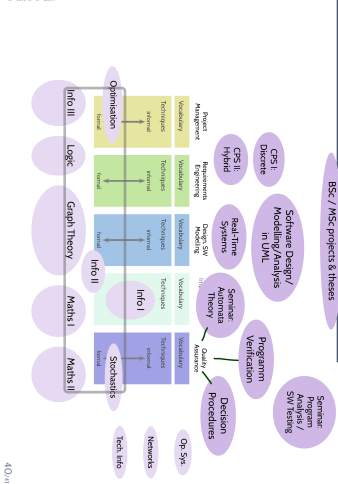
That's Today's Software Engineering—More or Less...

3714



Coming Soon to Your Local Lecture Hall...

Course Software-Engineering vs. Other Courses



Thursday, 2016-07-21, 12:00 to 14:00

Plenary Tutorial 6 & Questions Session
in 101-0-026 (right here)