Software Wartung und Evolution
Teil 4: Software Evolution

Harald Gall
Institut für Informatik
Universität Zürich
http://seal.ifi.unizh.ch

Universität Zürich

s.e.a.l. software evolution & architecture lab

Evolution: General Definition 1/2

- Evolution is the process of *progressive change* over time in characteristics, attributes, properties of some material or abstract, natural or artificial, entity or system or of a sequence of these
  - Changes are progressive when they result in a definable trend of, for example, increasing value, growing precision or better fit to a changing domain or context
  - Changes are not by chance, incidental, indeterministic, stochastic; there must be a trend

Evolution: General Definition 2/2

- Entities include objects or collections of objects (e.g. population) such as natural species, societies, cities, artefacts, concepts, theories, ideas or systems of these

- Change process will, in general, be continual with relatively slow rate of change, or discrete with individual incremental changes, small relative to entity as a whole

  - Source: [Lehman and Ramil 2001]

Software Evolution

- Keine genormte Definition
- Nach Lehman/Ramil
  - Software Evolution is the process of continual fixing, adaptation, enhancement to maintain stakeholder satisfaction
    - In response to changes in domains, needs, expectations
- Nach Bennet/Rajlich
  - Maintenance means general post-delivery activities
  - Evolution refers to a particular phase in the staged model where substantial changes are made to the software

Software Evolution

- Nach Godfrey
  - Evolution is what happens while you are busy making other plans
    - Maintenance is the *planned* set of tasks to effect changes
    - Evolution is what actually happens to software

Types of Programs

- *Nach Lehman, Belady 1980, pp. 1060-1076*
- S-type Programs („Specifiable“)
  - Problem can be stated formally and completely
  - Acceptance: Is the program correct according to its specification?
  - This software does not evolve
    - A change to the specification defines a new problem, hence a new program

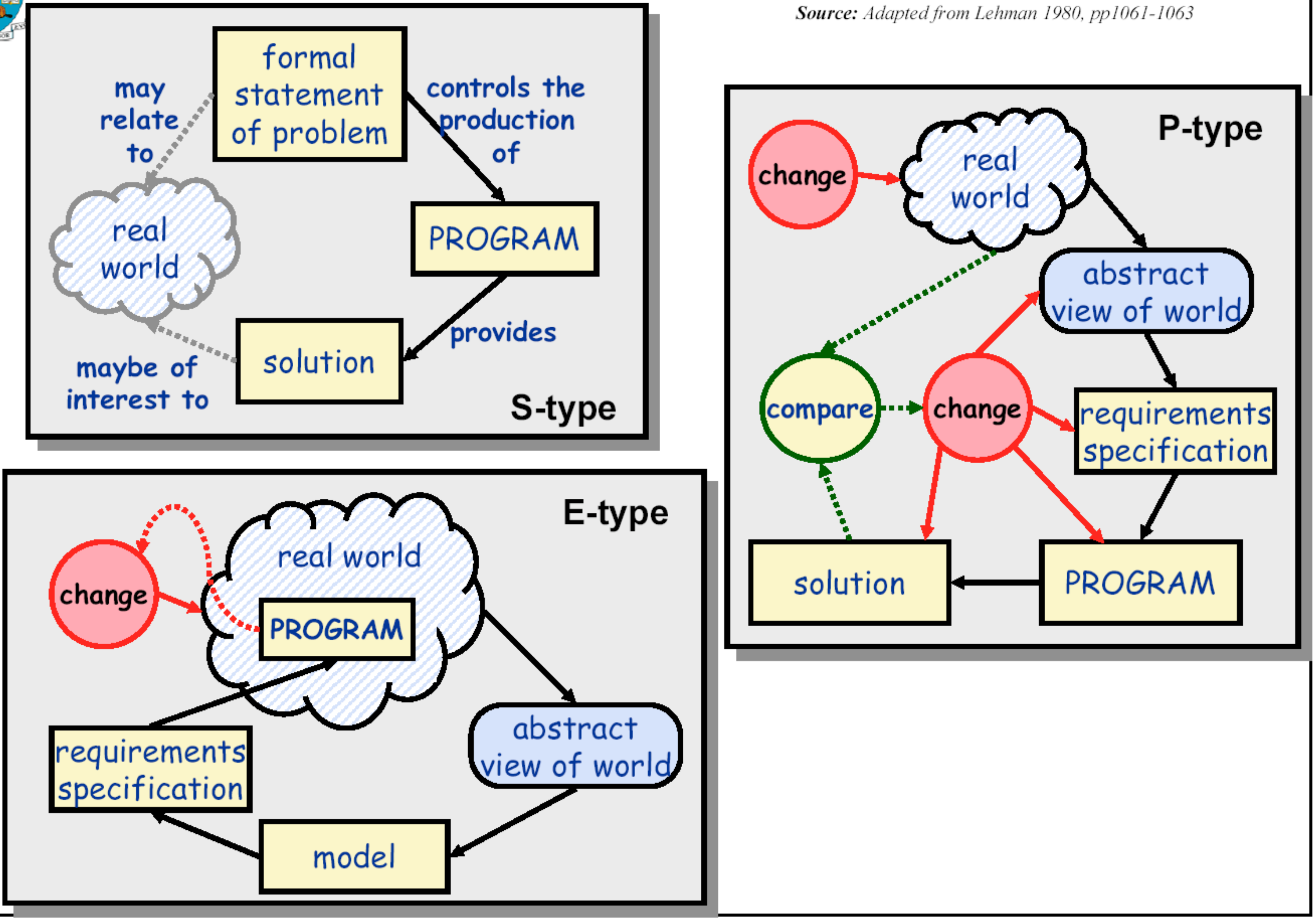
Types of Programs

- P-type Programs („Problem-solving“)
  - Imprecise statement of a real-world problem
  - Acceptance: Is the program an acceptable solution to the problem?
  - This software is likely to evolve continuously
    - Because solution is never perfect, and can be improved
    - Because the real-world changes and hence the problem changes

Types of Programs

- E-type Programs („Embedded“)
  - A system that becomes part of the world it models
  - Acceptance: Depends entirely on opinion and judgement; criterion is the satisfaction of stakeholder needs
  - This software is *inherently* evolutionary
    - Changes in the software and the world affect each other

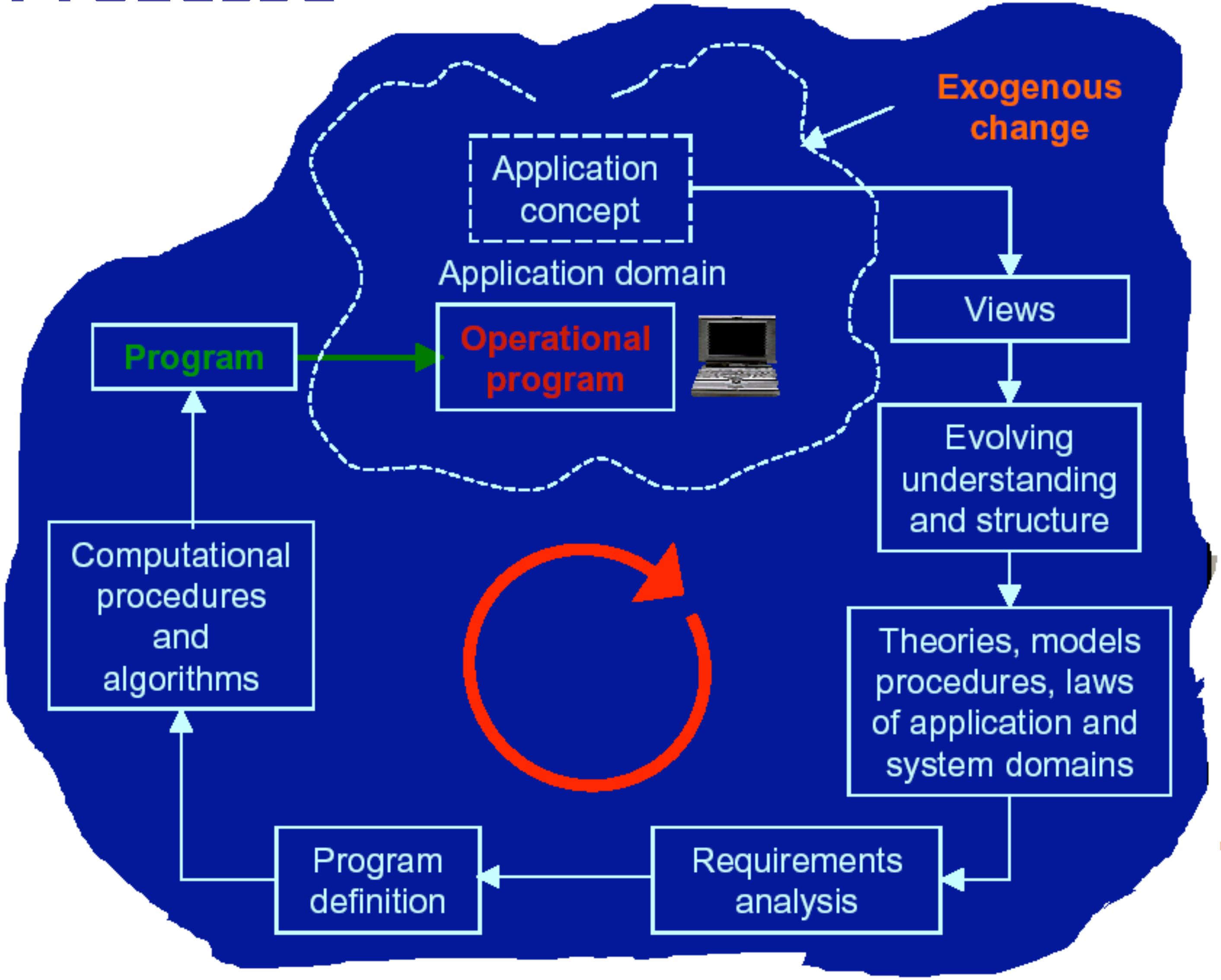
Source: *Adapted from Lehman 1980, pp1061-1063*

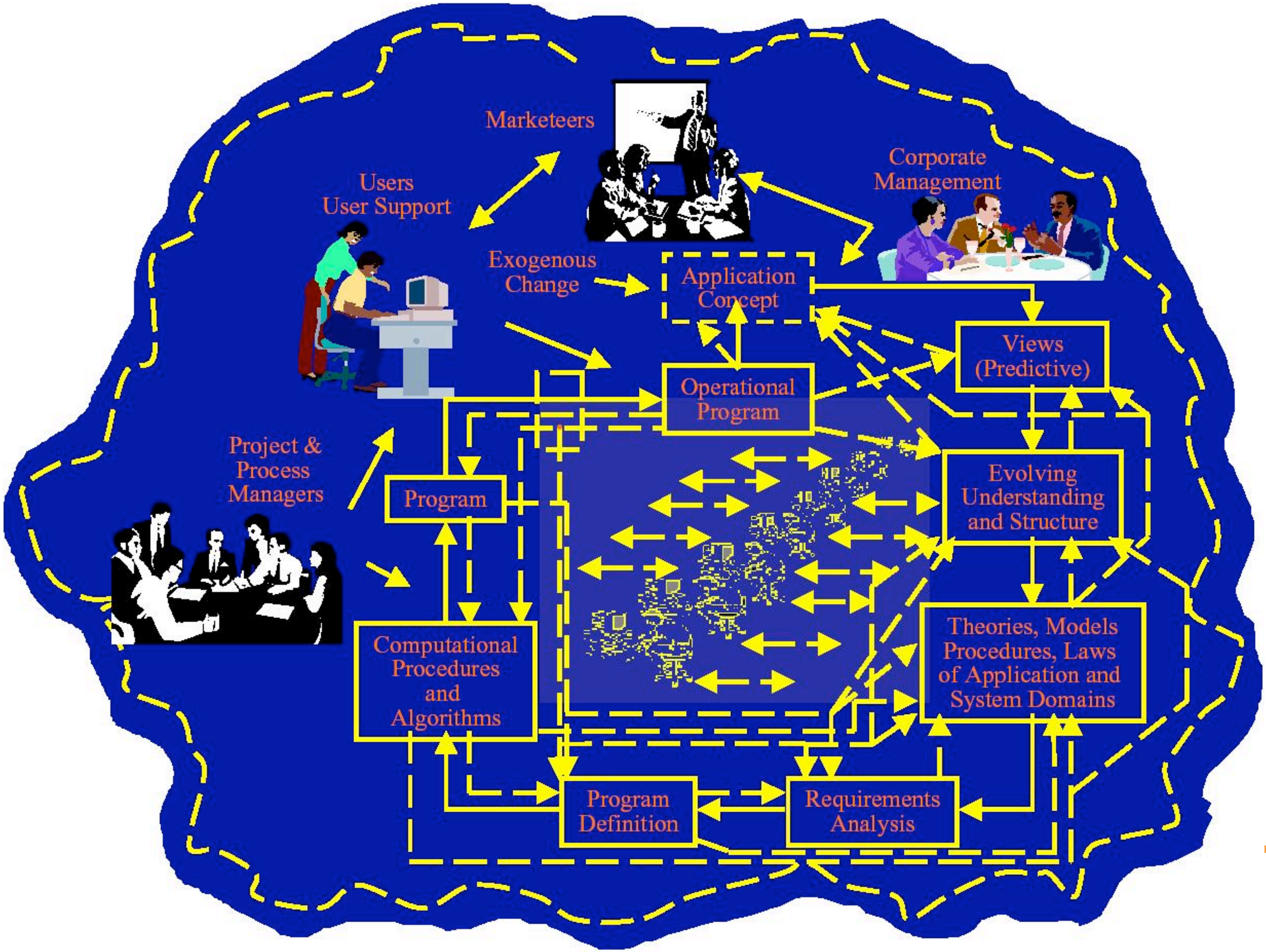


Software Systeme als Feedback Prozesse

- Der Entwicklungs- und Evolutionsprozess eines Software Systems wird von Lehman als
    - Multi-level
    - Multi-loop
    - Multi-agent
  - Feedback System bezeichnet.
- Feedback technisch: Die Rückführung eines Ausgangssignal als Eingangssignal in ein System
    - („Feedback: The return of a portion of the output, or processed portion of the output, of a (usually active) device to the input“)

Software Systeme als Feedback Prozesse

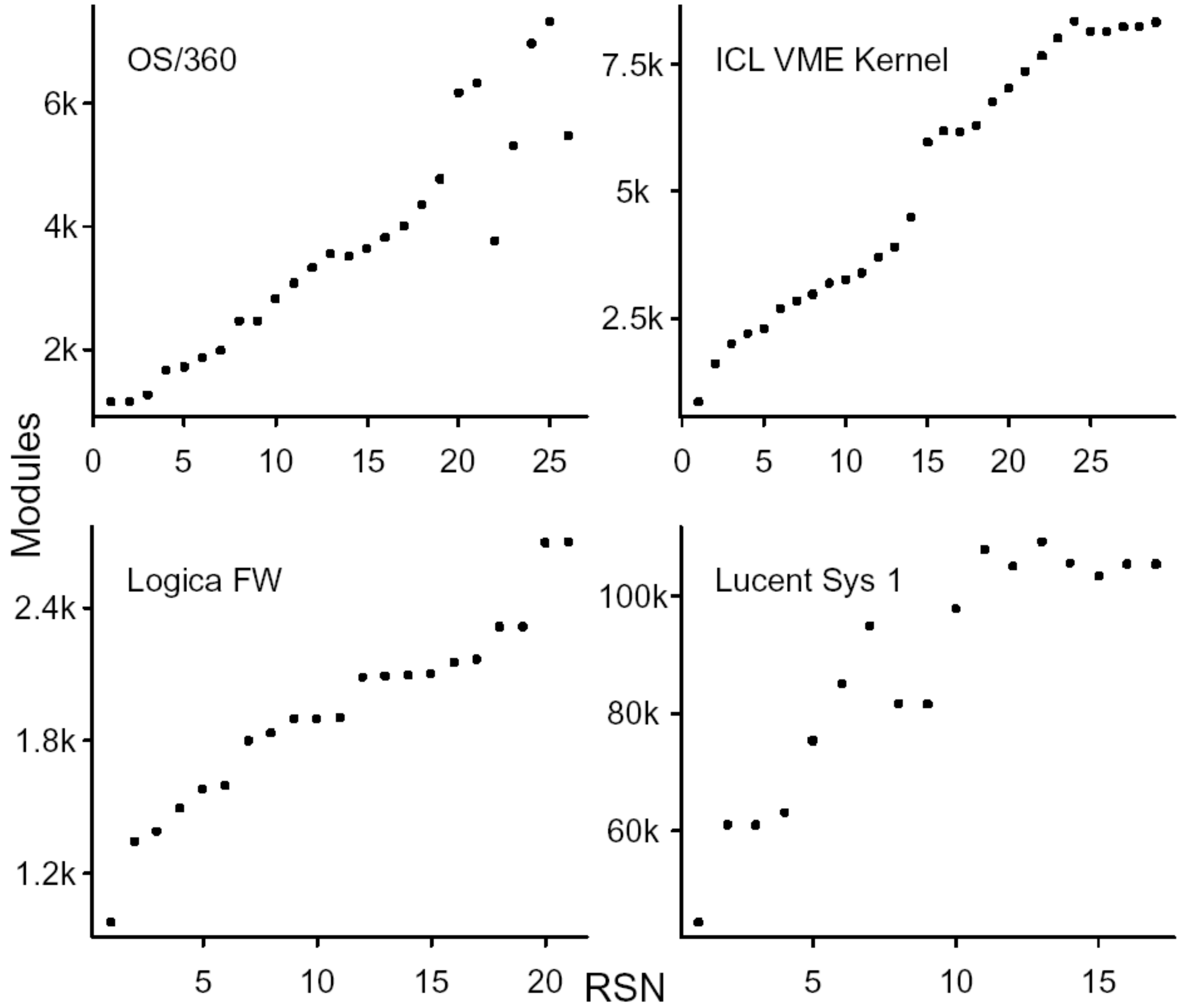



Marketeers
Users
User Support
Corporate
Management
Exogenous
Change
Application
Concept
Views
(Predictive)
Operational
Program
Project &
Process
Managers
Program
Evolving
Understanding
and Structure
Theories, Models
Procedures, Laws
of Application and
System Domains
Computational
Procedures
and
Algorithms
Program
Definition
Requirements
Analysis

Laws of Software Evolution

- In den späten sechziger Jahren untersuchen Lehman und Belady die Release History Daten von IBM OS/360 mittels bestimmter Metriken und stoßen auf Eigenschaften im Evolutionsprozess, die bei anderen Systemen in späteren Untersuchungen ebenfalls nachvollzogen werden können
- Diese Eigenschaften scheinen Gesetzmäßigkeiten zu folgen und wurden als „Laws of Software Evolution“ postuliert
- Die „Laws of Software Evolution“ ergeben sich aus der Beobachtung von E-type programs

Laws of Software Evolution



Laws of Software Evolution

- Warum „Gesetze“?
  - Die entdeckten Phänomene der Evolution werden als Gesetze bezeichnet, da sie technologie- und prozessunabhängige Mechanismen bezeichnen

Laws of Software Evolution

- *Nach Lehman, Belady 1980, pp. 1061-1063 und spätere Publikationen*
- (1) Law of continuing change
  - "A system that reflects some external reality undergoes continuing change or becomes progressively less useful
    - The change process continues until it becomes more economical to replace it by a new or restructured system."
- (2) Law of increasing entropy (or: complexity)
  - "The entropy of a system increases with time unless specific work is executed to maintain or reduce it."

Laws of Software Evolution

- (3) Fundamental law of software evolution
  - Software evolution is self-regulating with statistically determinable trends and invariants
- (4) Conservation of organisational stability (invariant work rate)
  - During the active live of a software system the *average effective global activity rate* is roughly constant

Laws of Software Evolution

- (5) Conservation of familiarity
  - In general, the average incremental growth rate (growth rate trend) tends to decline
    - As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery.
- (6) Continuing growth
  - The functional content of E-type systems must be continually increased to maintain user satisfaction

Laws of Software Evolution

- (7) Declining quality
  - The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes
- (8) Feedback System
  - E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base

Lehman's Approach: Formal

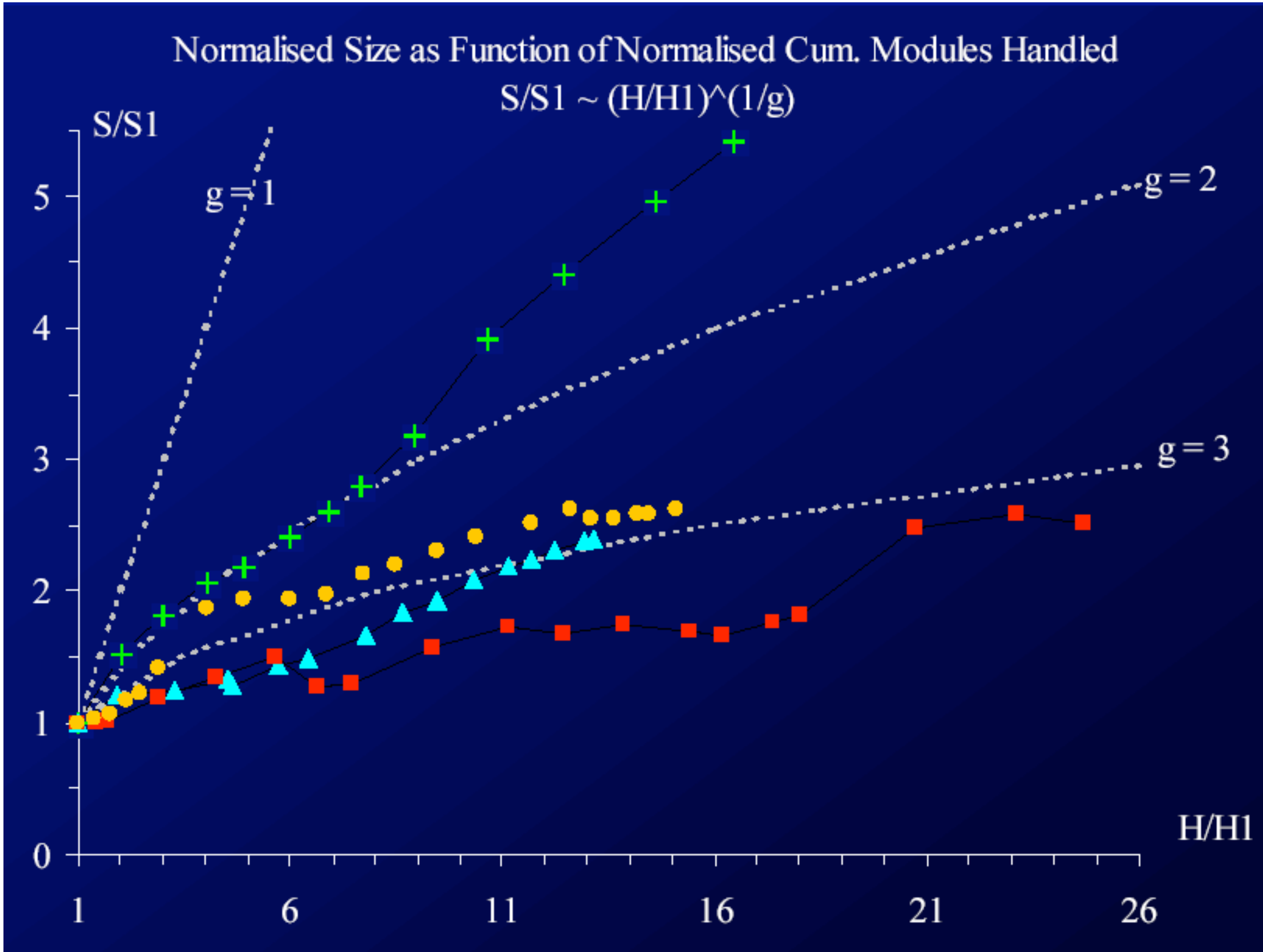
- Lehman describes software evolution on a formal level
  - Based on observations
    - Empirical generalisations are made
    - They provide basis for axioms in a formal theory
  - Possible inferences are proposed
    - Derived from the formal models
    - Basis for potential theorems in formal theory
  - Try to fully prove theorems

Formal Models of Software Evolution: Growth

- Inverse Square Model *[Turski 1996]*
  - $\underline{S}_1 = S_1$
  - $\underline{S}_i = \underline{S}_{i-1} + e / (\underline{S}_{i-1})^2$
    - *S ... Size (often number of modules)*
    - *i ... Release sequence number (1..n, n = max release nr.)*
    - *e ... Model parameter*
    - *S_i and $\underline{S}_i$ stand for actual and predicted size at release i*
- *Other model: Normalised size as a function of the normalised work rate [Lehman 2001]*
  - $\underline{S}_i / S_1 = (H_i / H_1)^{1/g'}$ *for i >= 1*
    - *H ... Work rate as indirect effort indicator (e.g. elements handled)*
    - *g´ ... Model parameter*

Formal Models: An Example

- Next slide shows the *normalised size as a function of the normalised work rate*
    - size measured in number of modules
    - work rate measure in modules handled
  - For four industrially evolved systems
  - Three different organisations
  - Three different application domains
- Data taken from release data history


Normalised Size as Function of Normalised Cum. Modules Handled
S/S1 ~ (H/H1)^(1/g)
S/S1
g = 1
g = 2
g = 3
H/H1
0
1
2
3
4
5
1
6
11
16
21
26

Formal Models: Use?

- Formal Models provide means for
  - Evolution planning
    - Simulation, visualisation, release planning
  - Process Management and Control
  - Long term prognosis
  - Overall process improvement
  - Tools

Research Areas in Software Evolution

- The driving force guiding the work will be the search for formally supported techniques:
  - logic-based declarative description and reasoning techniques
  - formal models for software evolution based on rewriting systems
  - software metrics
  - visualisation techniques
  - generation of design documents and source code
  - extraction of design and analysis documentation
  - migration to component-based and web-based systems
  - the use of meta-models as a general integration technique

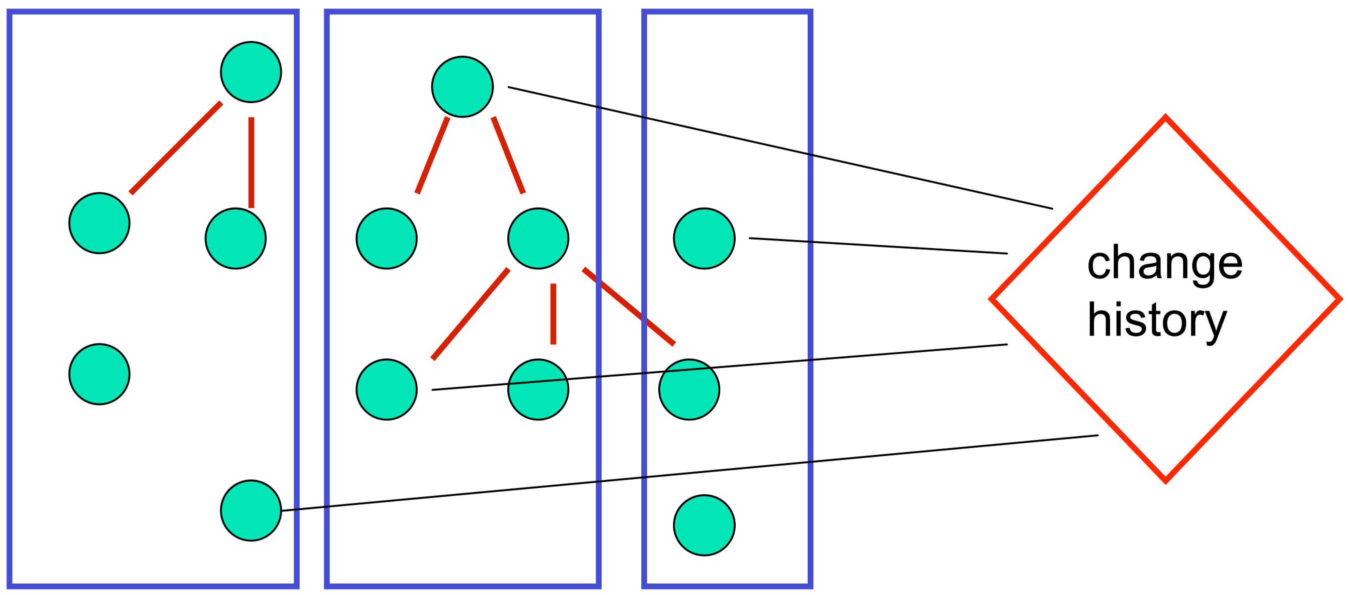
Analyzing Software Evolution Using Release History Data

http://seal.ifi.unizh.ch/projects/

What is Software Evolution Analysis?

- Investigating the evolution of a software system to identify potential shortcomings in its architecture or logical structure.
- Structural shortcomings can then be subject to reengineering or restructuring.

Motivation



Software Evolution Analysis

- **Goal**: learn from history about evolution of a system
- **Inputs**:
  - Version information (e.g. CVS, SVN, ClearCase)
  - Change information (author, date/time, size, messages, etc.)

- **Problems**:
  - Reveal common change behavior of classes
  - Identify logical coupling among classes
  - Evaluate modules and the entire system
  - Identify spots of design erosion, architectural decay, etc.

The QCR-approach

- **Q**uantitative Analysis (QA)
  - analyzes the change and growth rates of modules (classes) across releases and provides outliers

[Gall et al. 97]
[Jazayeri 02]

- **C**hange Sequence Analysis (CSA)
  - identifies common change history of modules and provides structural dependencies based on common change sequences (e.g. <1,3,5,6,7,8>)

[Hajek, Gall, Jazayeri 98]
[Riva, Gall, Jazayeri 99]

- **R**elation Analysis (RA)
  - compares modules (classes) based on CVS change history information and reveals module dependencies, ie. logical couplings
  - *units of interest* = *classes*
  - *change information* = *CVS data*

[Gall, Krajewski 03,06,07]

Quantitative Analysis

Analyzing quantitative aspects of software evolution

s.e.a.l.
software evolution & architecture lab

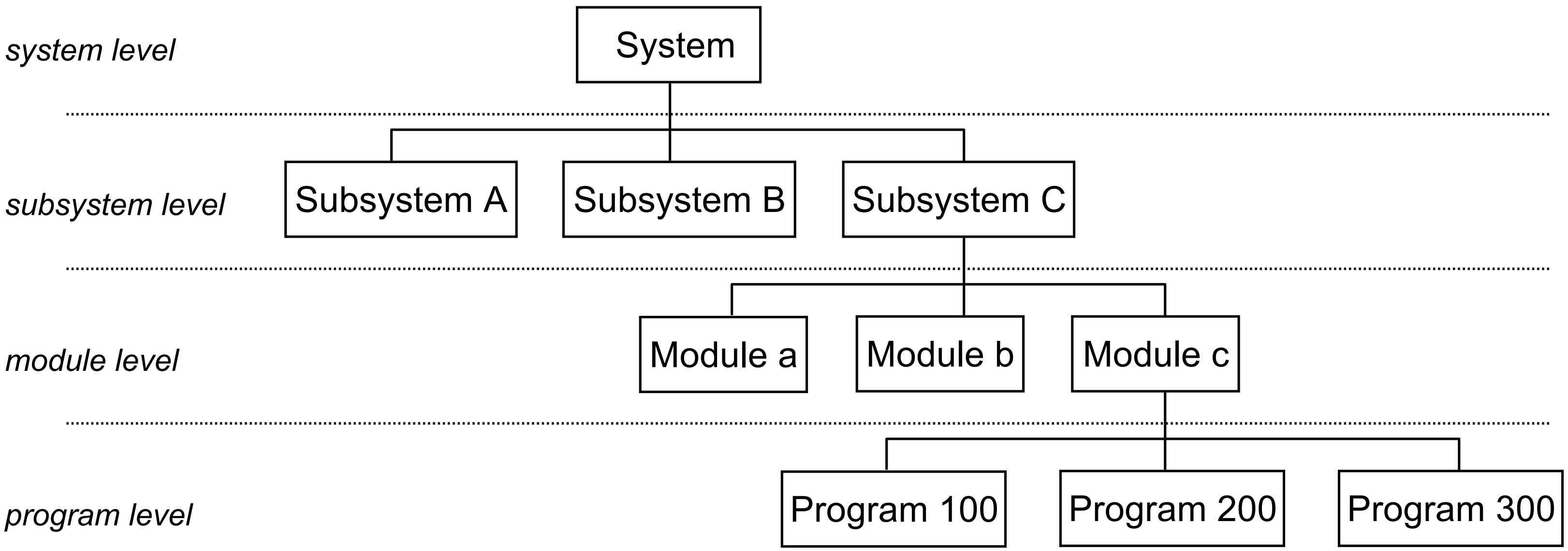
Quantitative Evolution Analysis

- Goals:
  - Identify potential shortcomings of a Telecommunication Switching System (TSS) by tracking its historical development
  - Use database containing structural information about 20 releases of the TSS delivered over a period of 2 years
  - Focus on macro-level:
    - investigate only structural information about each release (version numbers)
    - no source code metrics at all

QA: Approach

- Approach:
  - Observe software evolution via release history
  - Detect logical coupling via
    - Change Sequence Analysis, and
    - Change Report Analysis
  - Visualize software release histories using color and third dimension

Telecommunication Switching System (TSS)



The Release Database

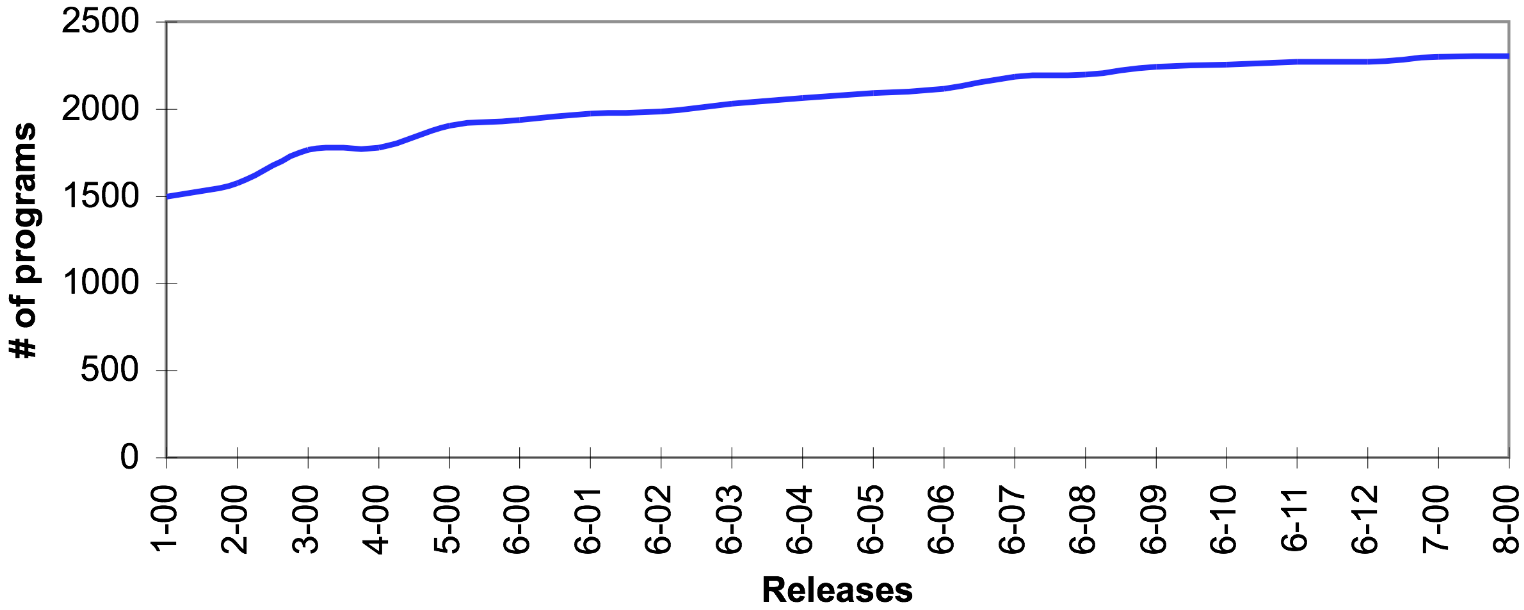
- For each release stored:
  - Entries for elements at system, subsystem, module, and program level together with relations among them
  - Systems and programs are characterized by version numbers
  - Program version numbers are independent of the system's version number
  - Changes result in incremented version number(s)

- Each system release consists of
  - 8 subsystems, 47 to 49 modules, and 1500 to 2300 programs.

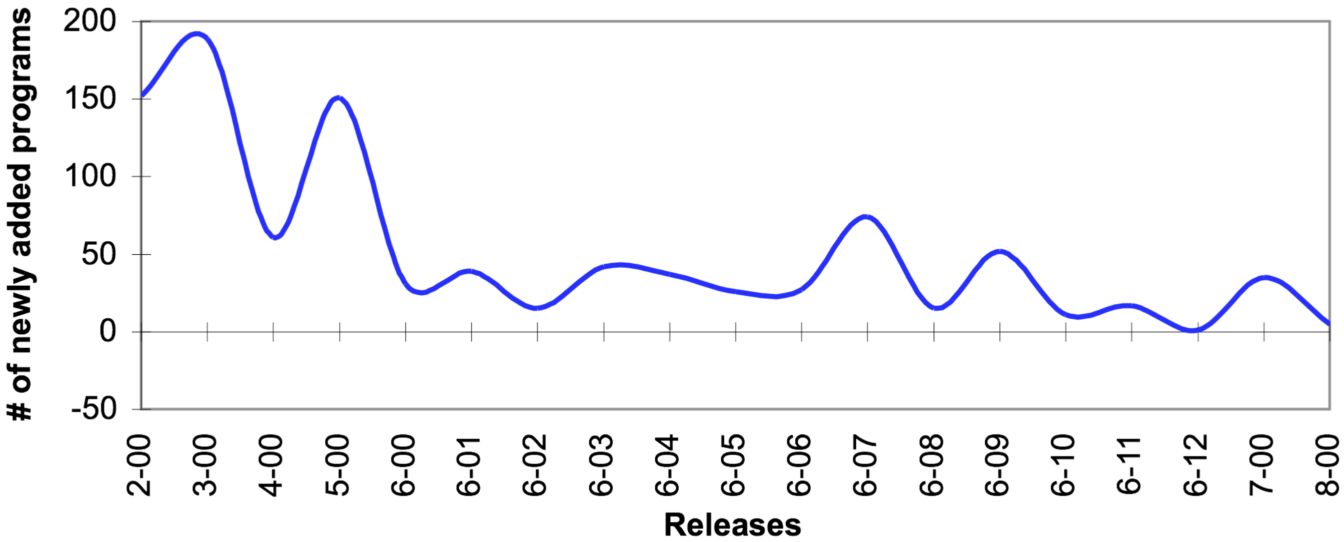
Software Evolution Observations

- Problem: extract useful information from the PRDB to reason about the evolution of the TSS
- We focus on the following system properties:
  - Size of system, subsystem, or module: number of programs (as the module "unit")
  - Change rate: percentage of programs (identified by a different version number)
  - Growth rate: percentage of programs added (or deleted) from one release to the next
- Evolution of *whole* system and *particular* subsystems

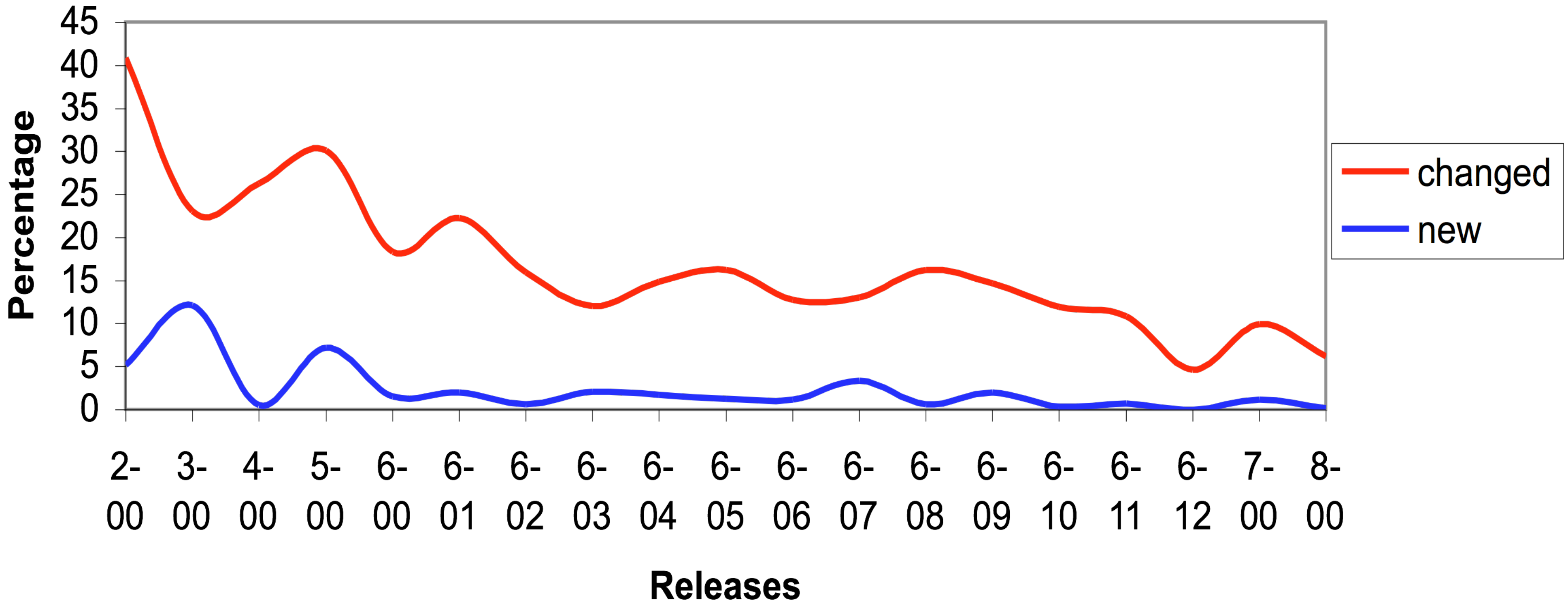
TSS: System size over time



TSS: Newly added programs per release



TSS: Change and growth rates

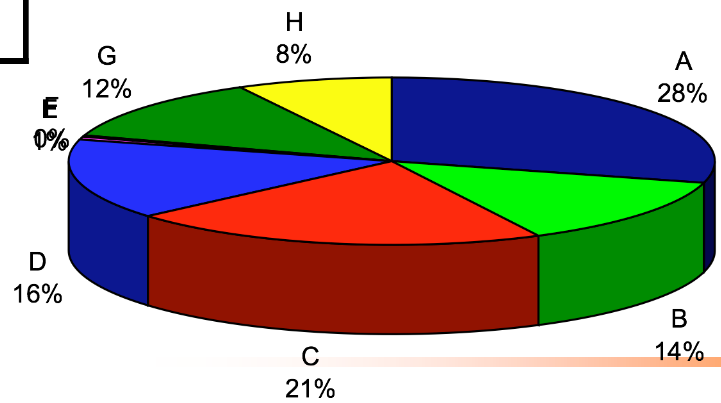


TSS: System Observations

- High growth rate
  - Increased from initially 1499 programs to 2303 at the end
  - 53% increase in 21 months
- Findings
  - The size of the system is growing linearly
  - Between 2.00 and 5.00 and in 7.00 major activities
  - Only a few added programs in 8.00
  - The structure of the whole system has become stable
- Evolution in a satisfactory way?

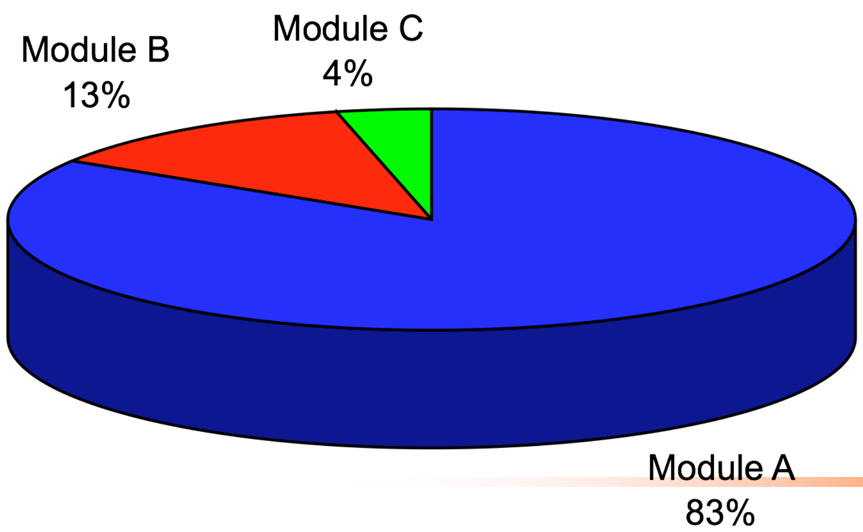
TSS Subsystems / Change and growth

Subsystem	Change rate (%)	Growth rate (%)
A	11	18
B	16	18
C	25	193
D	5	78
E	8	8
F	33	-25
G	29	157
H	20	3

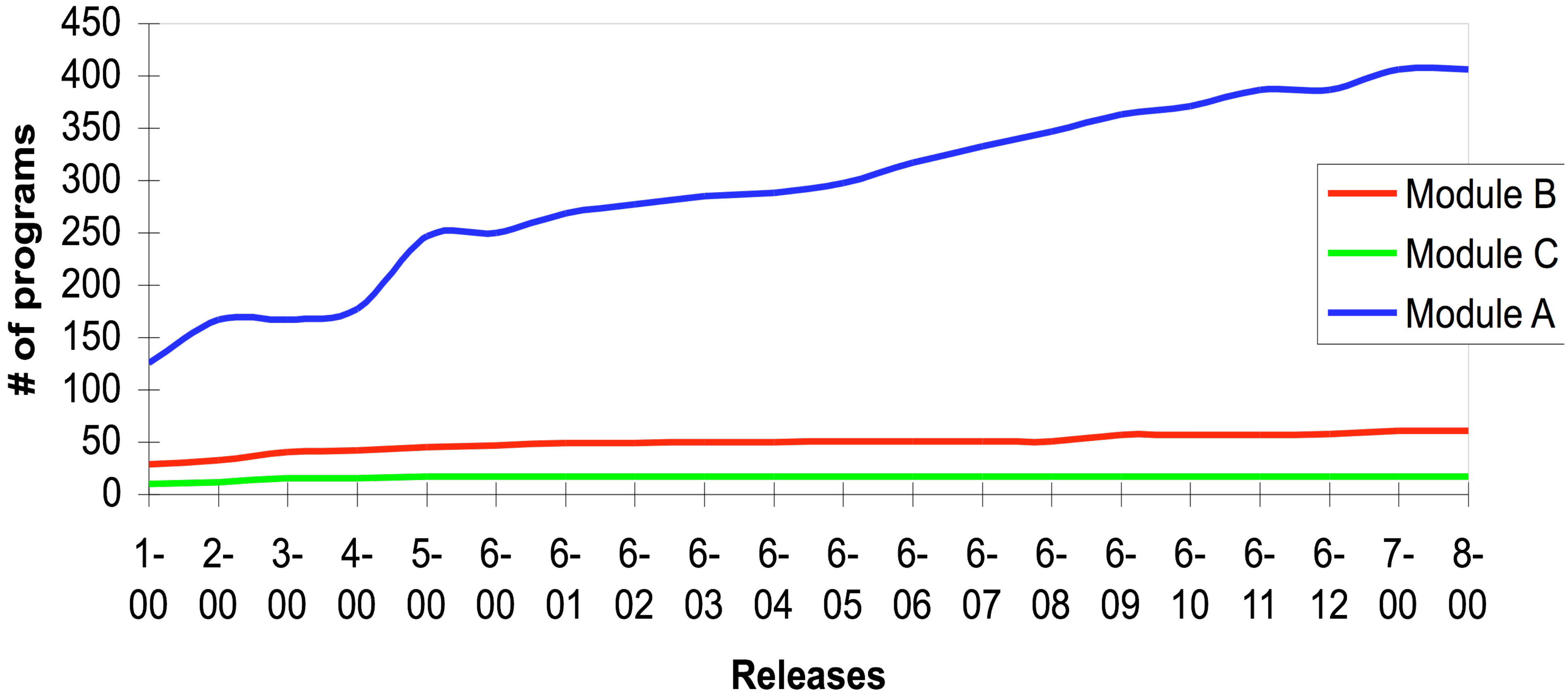


Evolution of Subsystem C

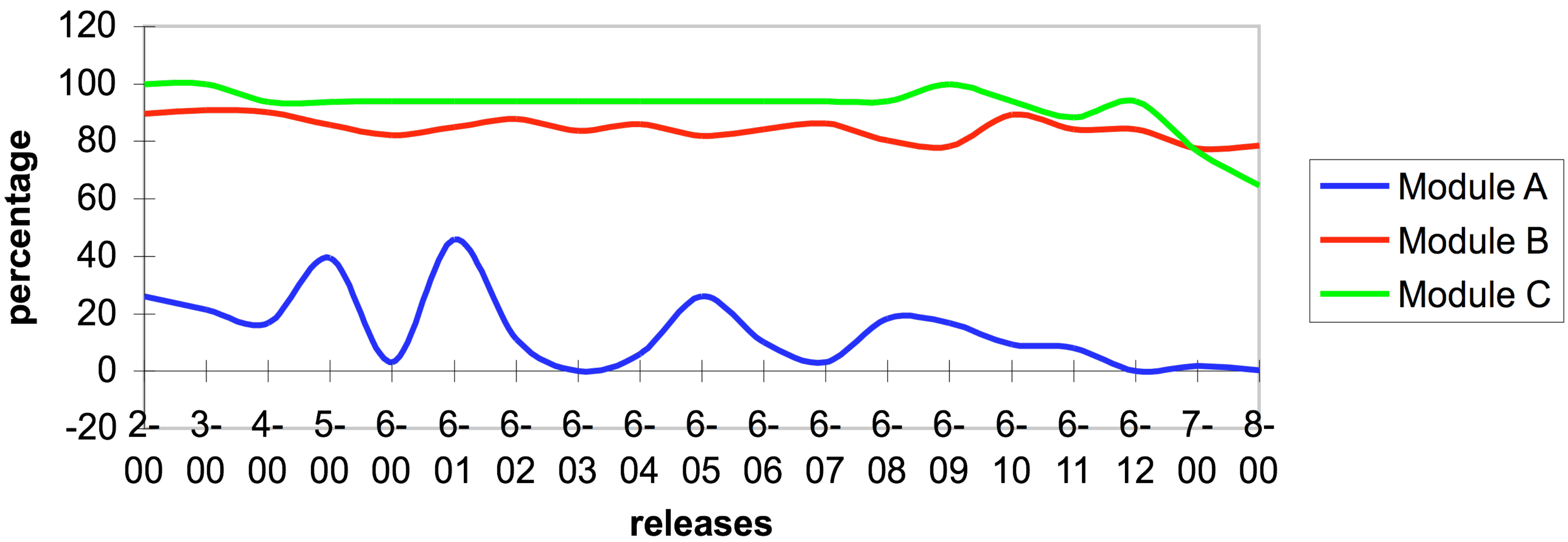
- Characteristics of Subsystem C:
  - highest growth rate and
  - one of the highest change rates among all subsystems
  - therefore most likely candidate for restructuring



Subsystem C: Sizes of modules



Subsystem C: Change rates of modules



Interpretation of Data

- Development of the whole system:
  - becomes stable over the twenty releases
  - change and growth rates decrease as do the number of added programs per release
  - structure seems fine
- Development of subsystems (Subsystem C):
  - the picture changes significantly!
  - high growth and change rate (Modules B & C)
  - similar names with only different endings
    - ➔ copying and slightly modifying!

Change Sequence Analysis

Detection of Logical Coupling

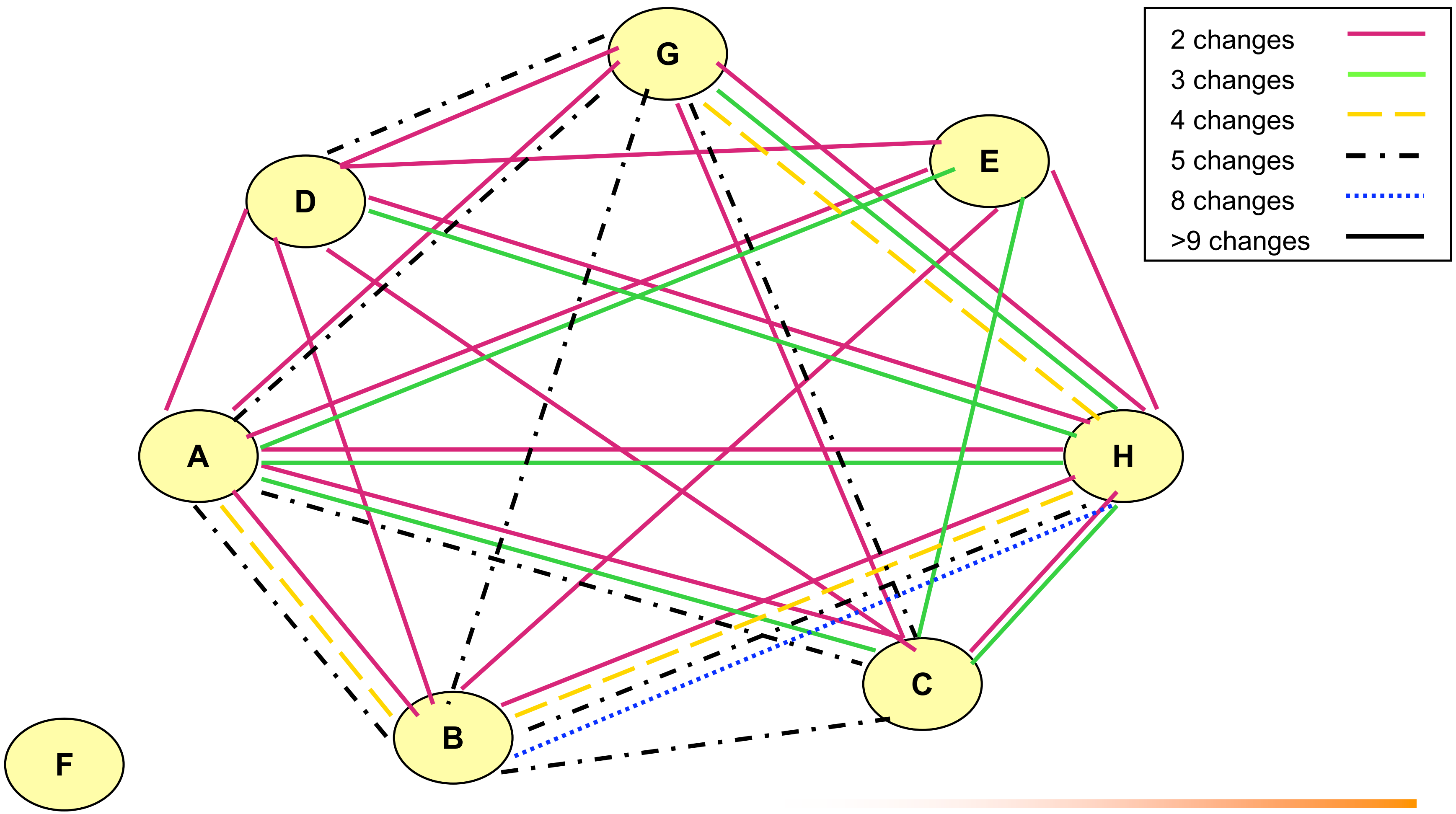
Detection of Logical Coupling

- Change Sequence of a program <1 2 3 5 7 11>
  - program changed in releases 1, 2, 3, 5, 7, and 11
  - 5 changes
- Subsequences as contiguous parts
  - <1 2 3>, <3 5 7>, etc.
- Changes are represented by a (sub-) sequence
- Identify potential “logical couplings” among programs

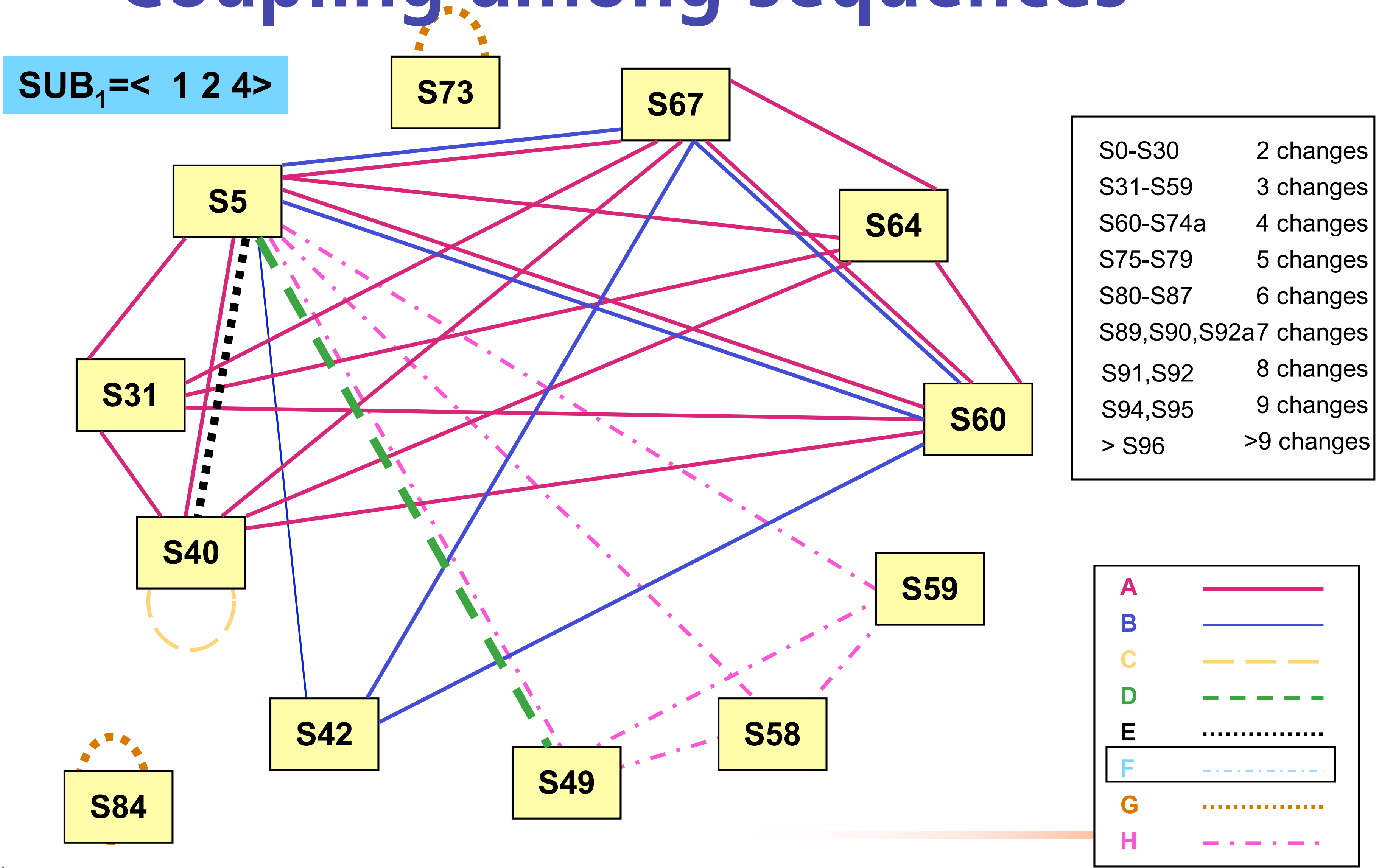
Change Sequence Analysis

- Approach:
  - compare change sequences of different modules
  - identify patterns of change
  - identify common “change sequences” (patterns)
- Result: potential logical couplings

Coupling among subsystems



Coupling among sequences



Change Report Analysis

- Goal / Approach:
  - verify logical coupling
  - examine change reports of modules with the same change sequence
  - same reason for change defines logical coupling
- Result: logical couplings among modules / subsystems

Example of a change report

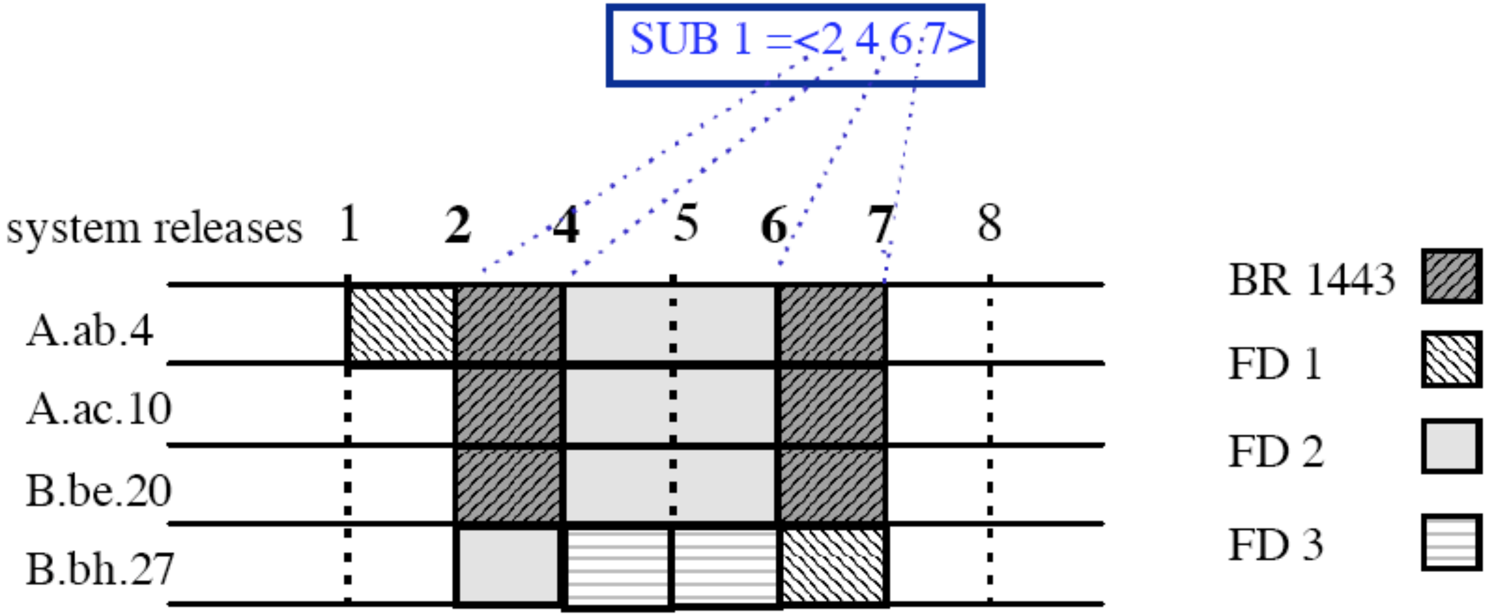
Ver 2.4 — 96/03/12 10:10:07
TSS---PROGRAM CHANGE DESCRIPTION

ELEMENT NAME: Program 111 **2.3 --> 2.4**
CHANGED BY: John DOE
CHANGES as follows:

CHANGE NR: 1
CHANGE TYPE: B // **bug fix**
REFERENCE: BR 1443 // reference to a **bug report number**
ERROR CLASS: A // **error class**, i.e. operation in working state
DESCRIPTION: hanging of the circuits in environment xy.

CHANGE NR: 2
...

Change Reports Analysis



Résumé

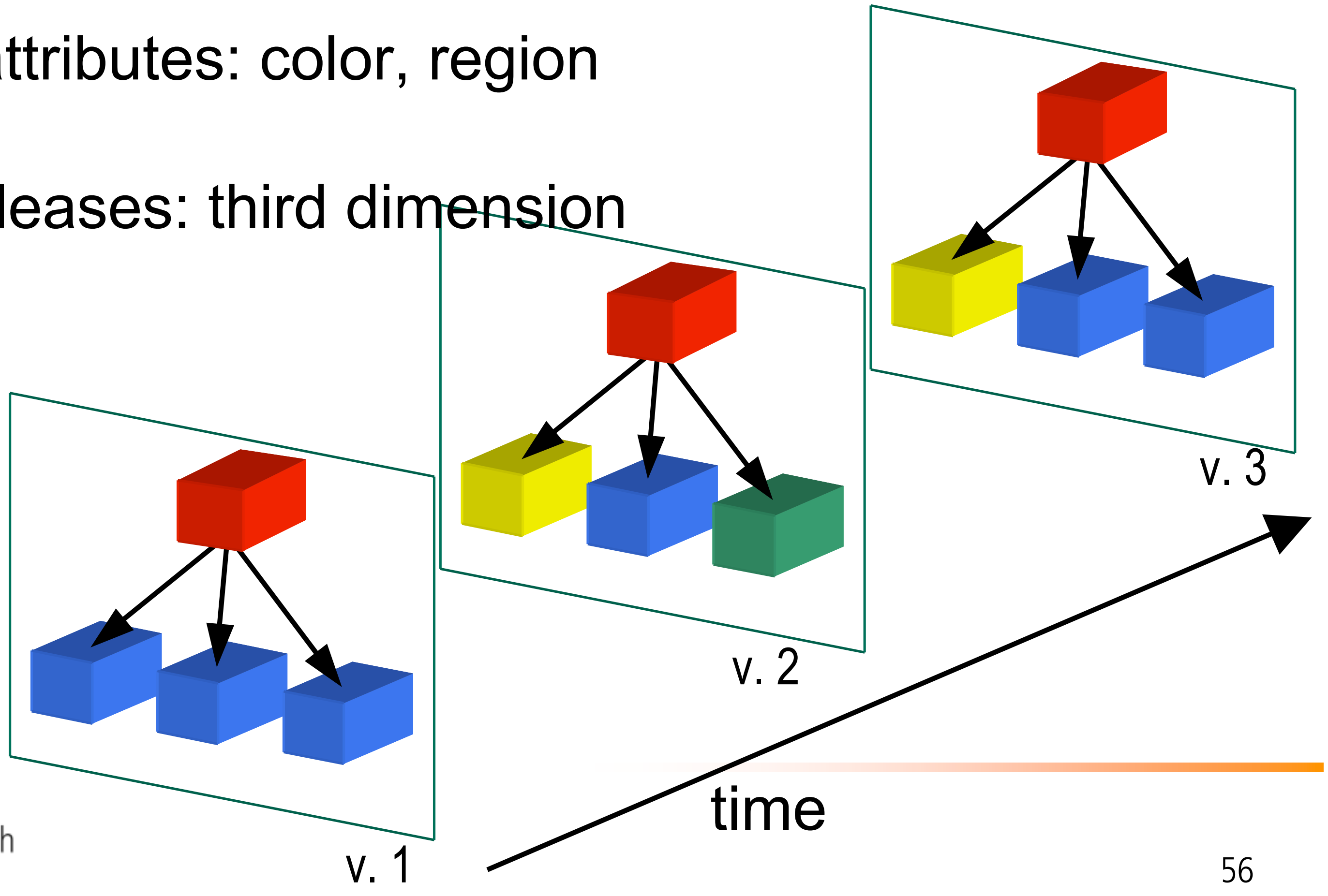
- Identified modules and programs that should undergo restructuring / reengineering
- Detected potential logical coupling via change sequences
  - Stronger logical couplings via longer sequences
- Verified logical coupling via change reports

Visualizing Release Histories

s.e.a.l.
software evolution & architecture lab

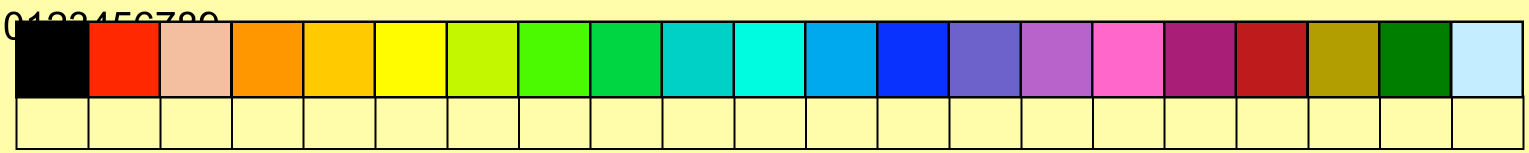
Visualization

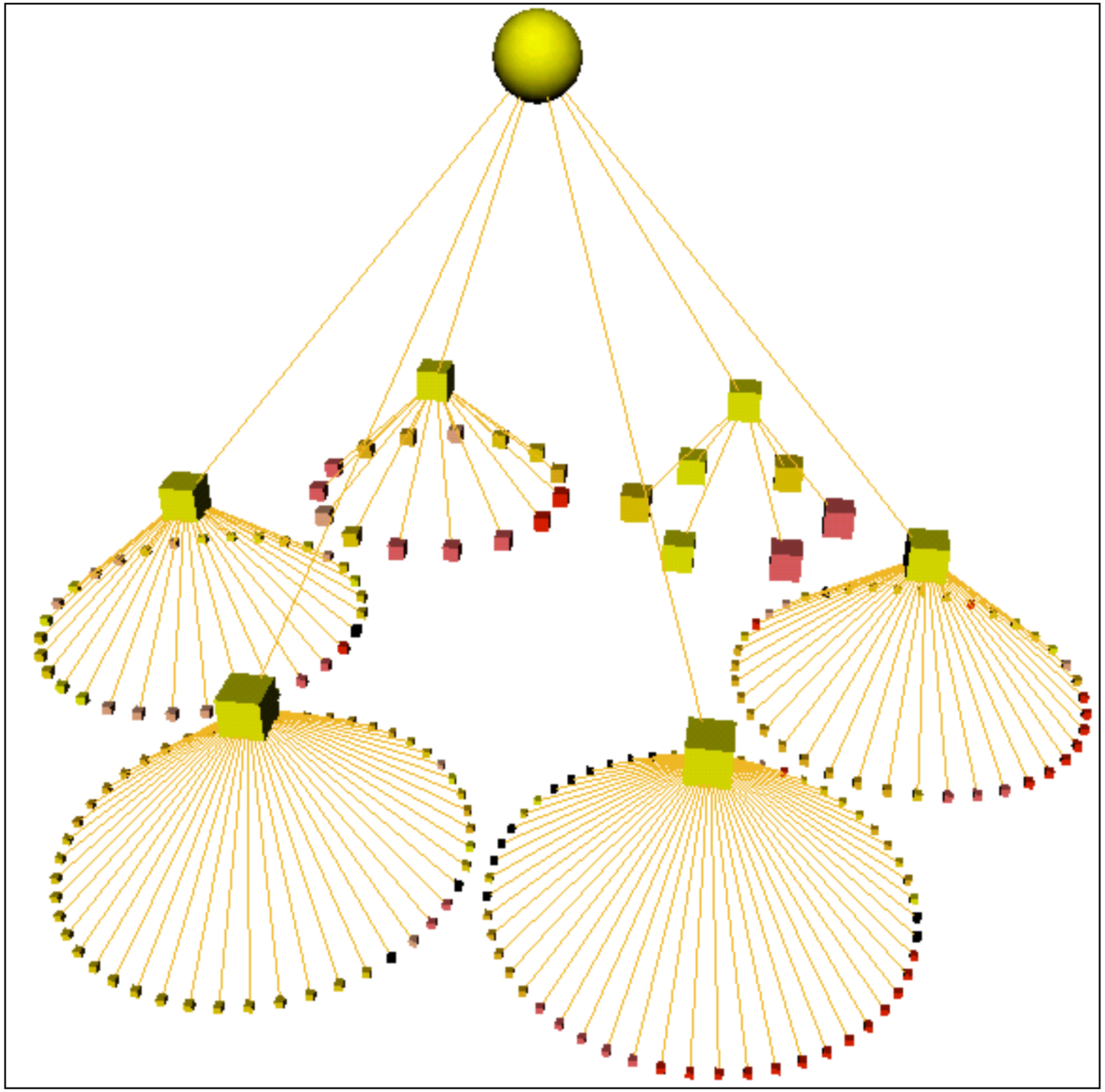
- Structure of the system: visualization of tree structure (2-D and 3-D)
- Software attributes: color, region filling
- Multiple releases: third dimension

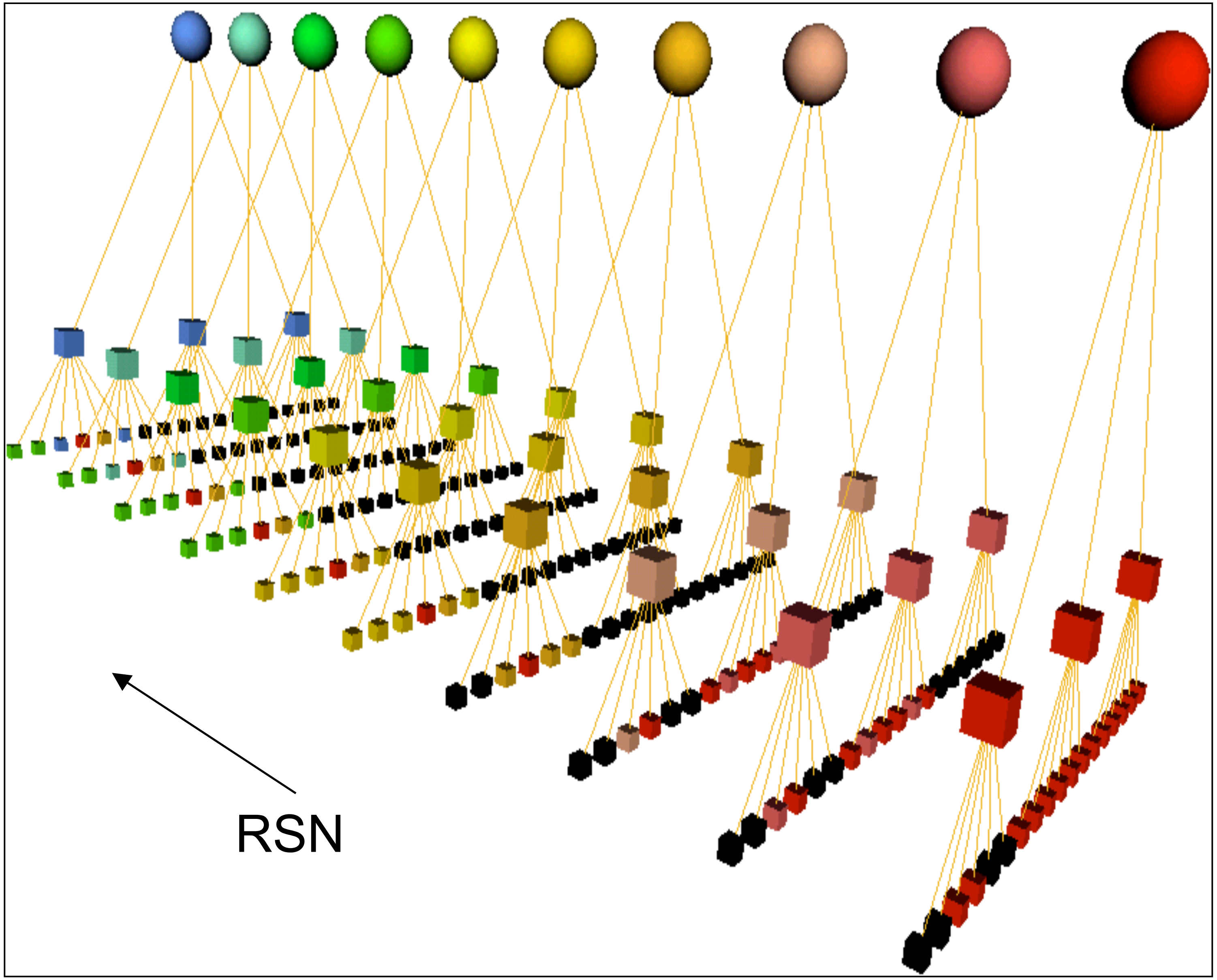


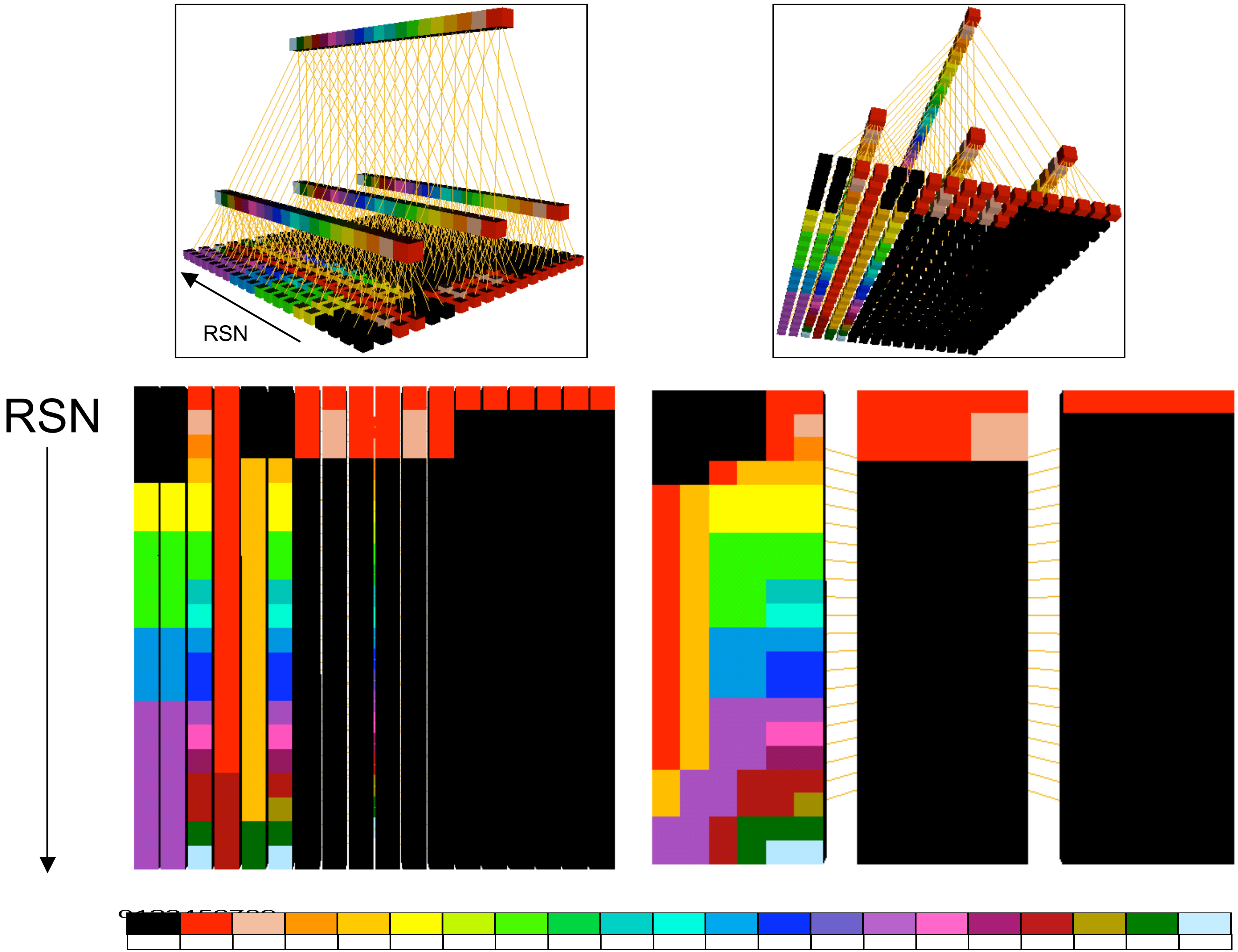
The Database & Color Scale

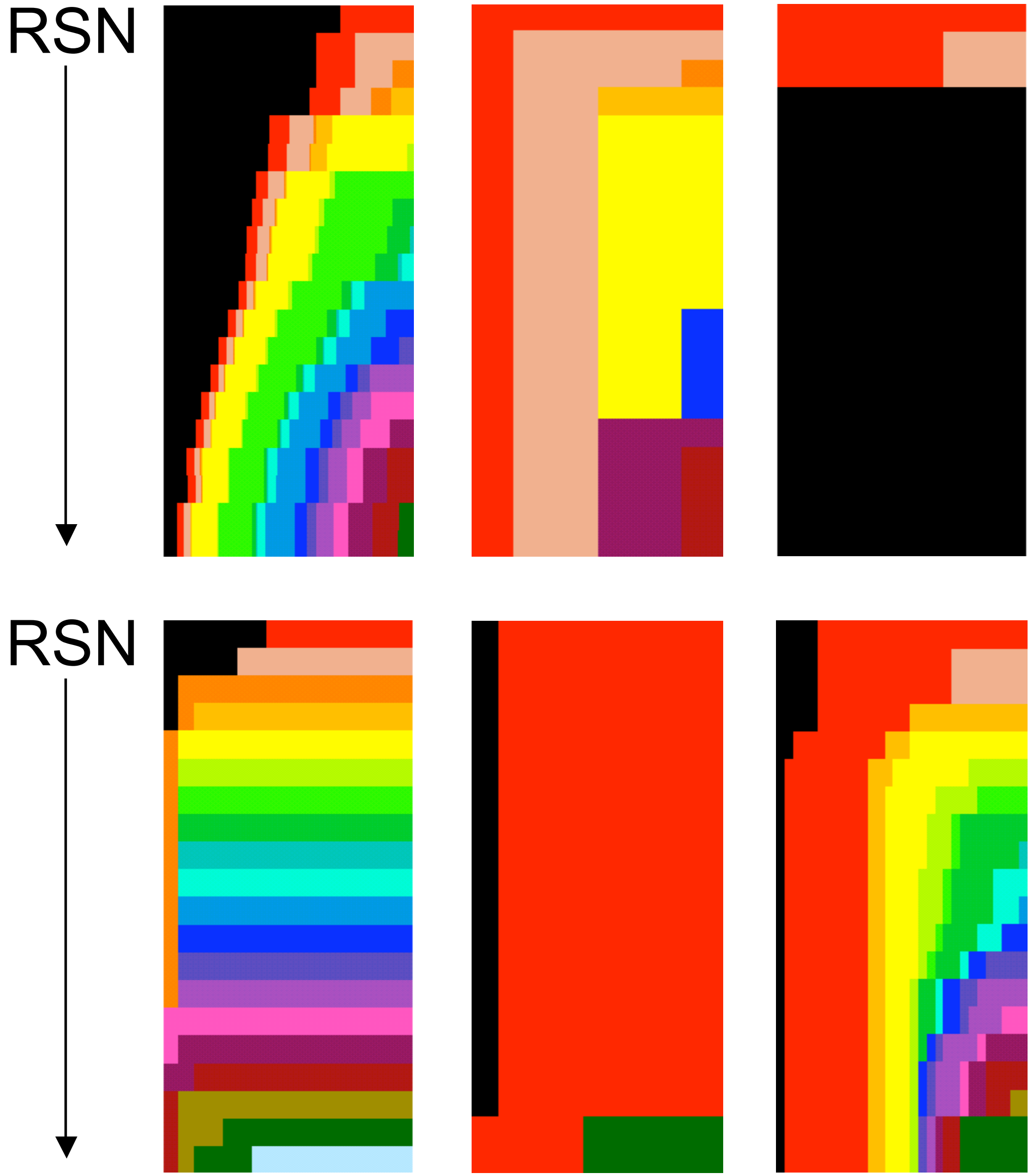
Subsystem Level	Module Level	Program Level	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
AAA	BBB	AAAPRG	0	0	0	0	5	5	7	7	7	7	11	11	11	14	14	14	14	14	14	14
AAA	BBB	BBBPRG	0	0	0	0	5	5	7	7	7	7	11	11	11	14	14	14	14	14	14	14
AAA	BBB	CCCPRG	1	2	3	4	5	5	7	7	9	10	11	12	12	14	15	16	17	17	19	20
AAA	BBB	DDDPRG	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	17	17	17	17
AAA	BBB	EEEPRG	0	0	0	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	19	19
AAA	BBB	FFFPRG	0	0	0	4	5	5	7	7	9	10	11	12	12	14	15	16	17	18	19	20
AAA	CCC	000PRG	1	1	1																	
AAA	CCC	222PRG	1	2	2																	
AAA	CCC	333PRG	1	1	1																	
AAA	CCC	444PRG	1	1	1																	
AAA	CCC	555PRG	1	2	2																	
AAA	CCC	666PRG	1	1	1																	
AAA	DDD	XXXXPRG	1																			
AAA	DDD	YYYYPRG	1																			
AAA	DDD	ZZZZPRG	1																			
AAA	DDD	KKKKPRG	1																			
AAA	DDD	JJJJPRG	1																			
AAA	DDD	LLLLPRG	1																			



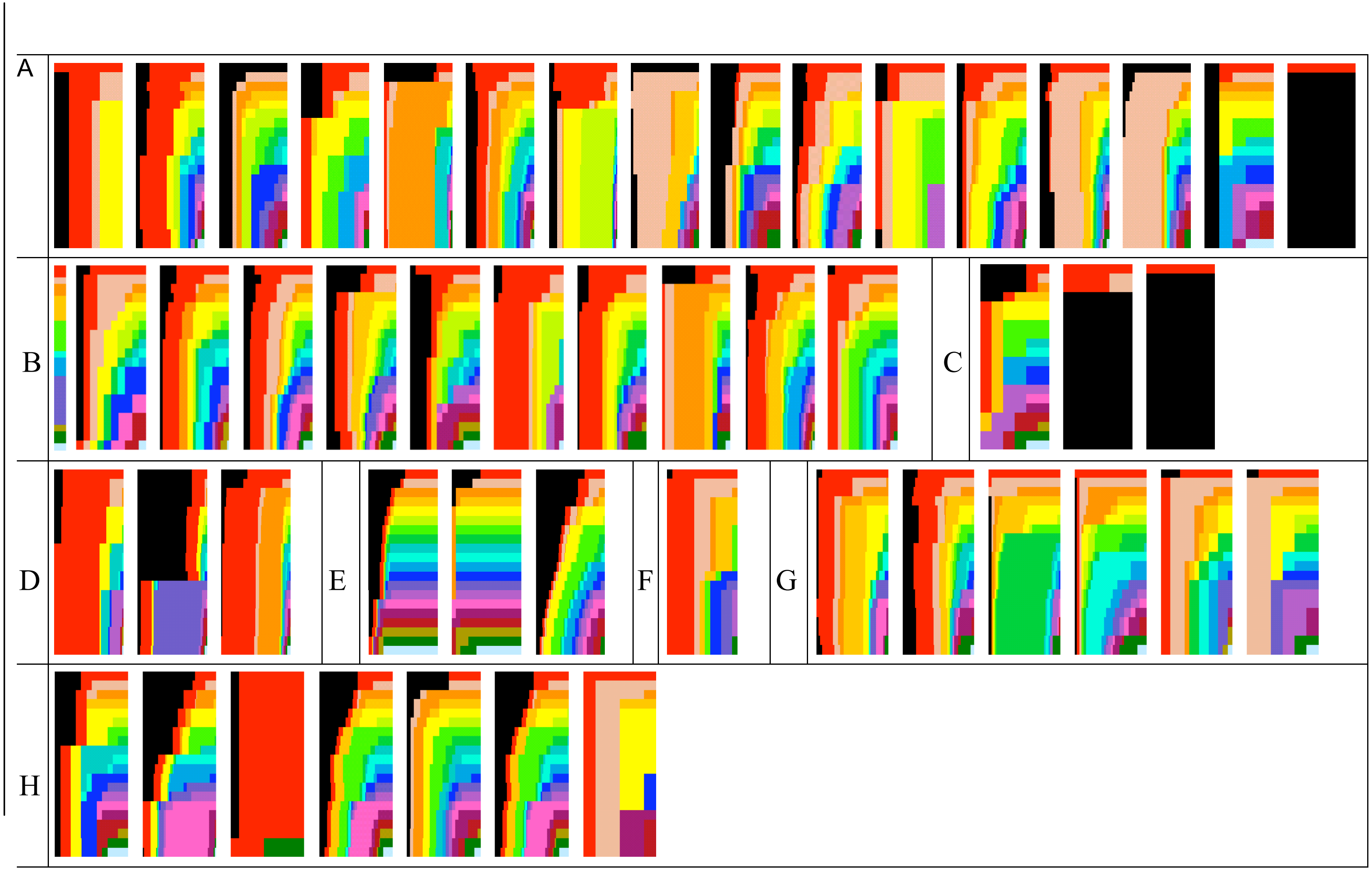



RSN


RSN
RSN


RSN
RSN

TSS visualized



Relation Analysis

Finding Class Couplings via
Change Dependencies

Relation Analysis (RA)

- ideally, components could be changed independently of each other, but …

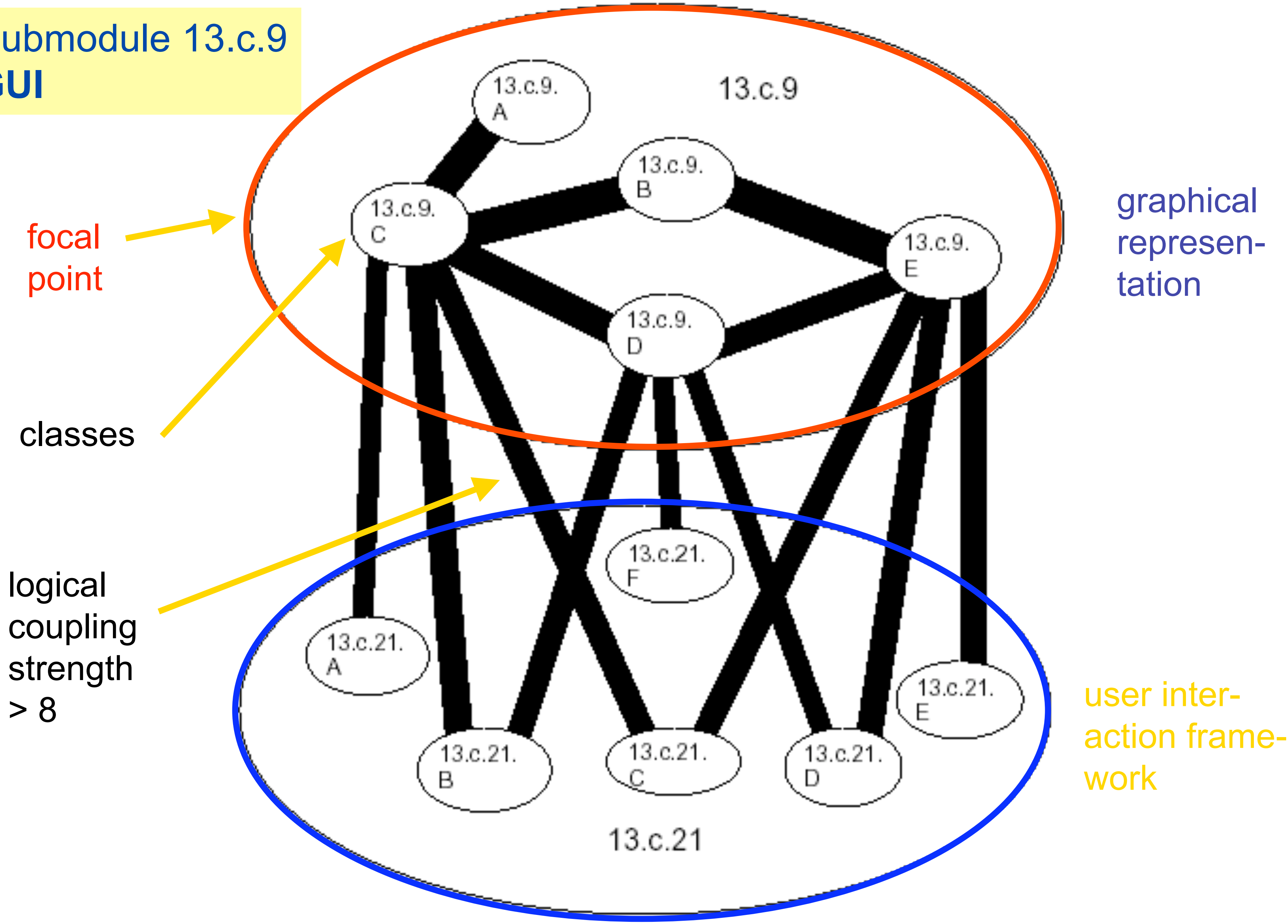
- evolution of classes is compared to identify those that were most frequently changed together
- comparison is based on author name, date and time of the check-in of a particular change
- based on the strong code ownership in case study and time window (4 minutes) for check-in

- as a result RA reveals logical coupling
- the number of common changes = strength of the logical coupling

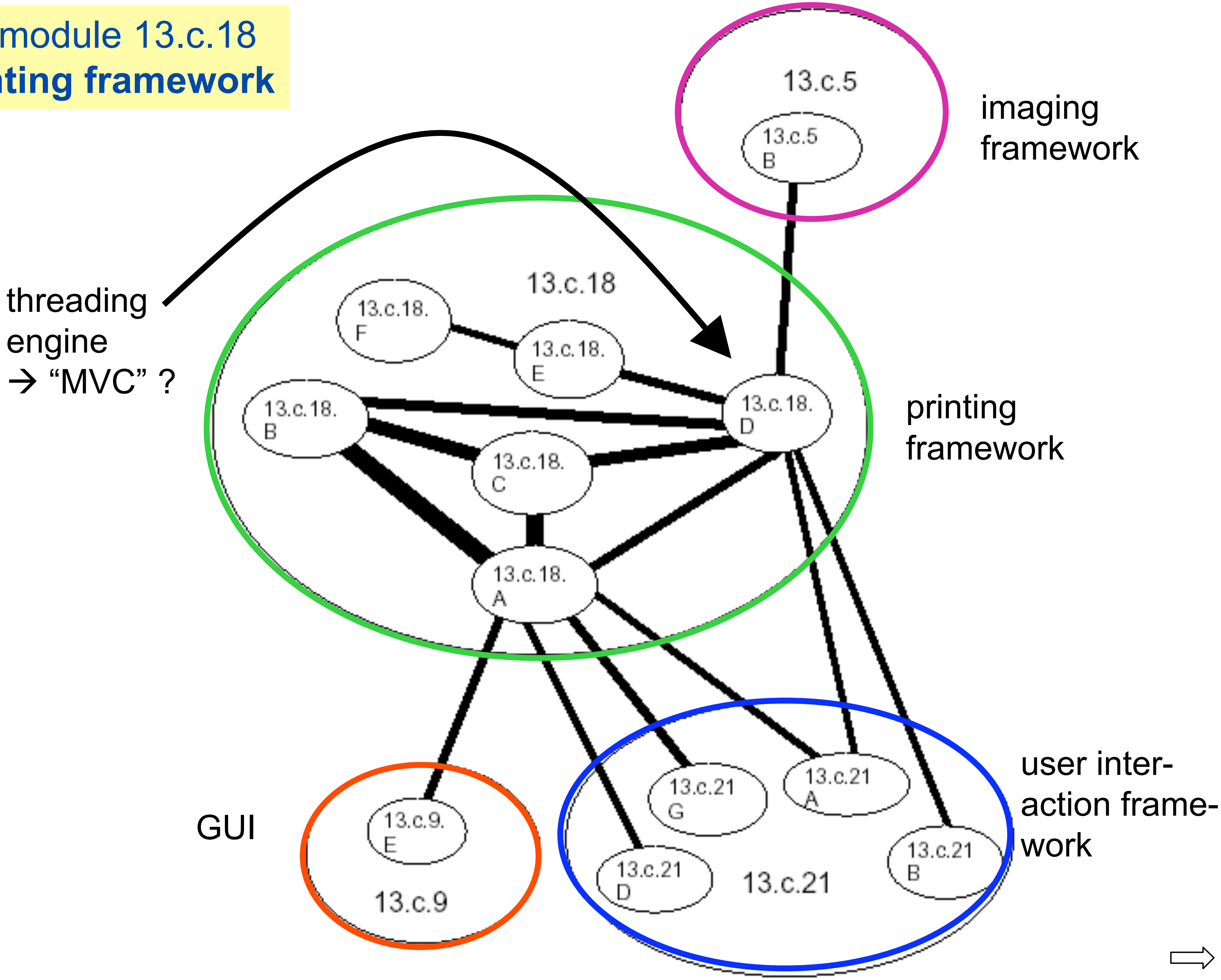
Case study: PACS

- A Picture Archiving and Communication System (PACS)
  - mission: medical pictures that doctors can view and archive for diagnosis; supporting different kinds of workstations:
    - viewing-only
    - additional diagnostic features allowing to change attributes of images, mark particular regions, sort and arrange pictures in sequences, annotate with information, etc.
  - implemented in Java, 5.500 classes (~500.000 LOC)
  - configuration files
  - analysis period: April 2000 – July 2002, ie. 28 months
  - vendor wants to support product families

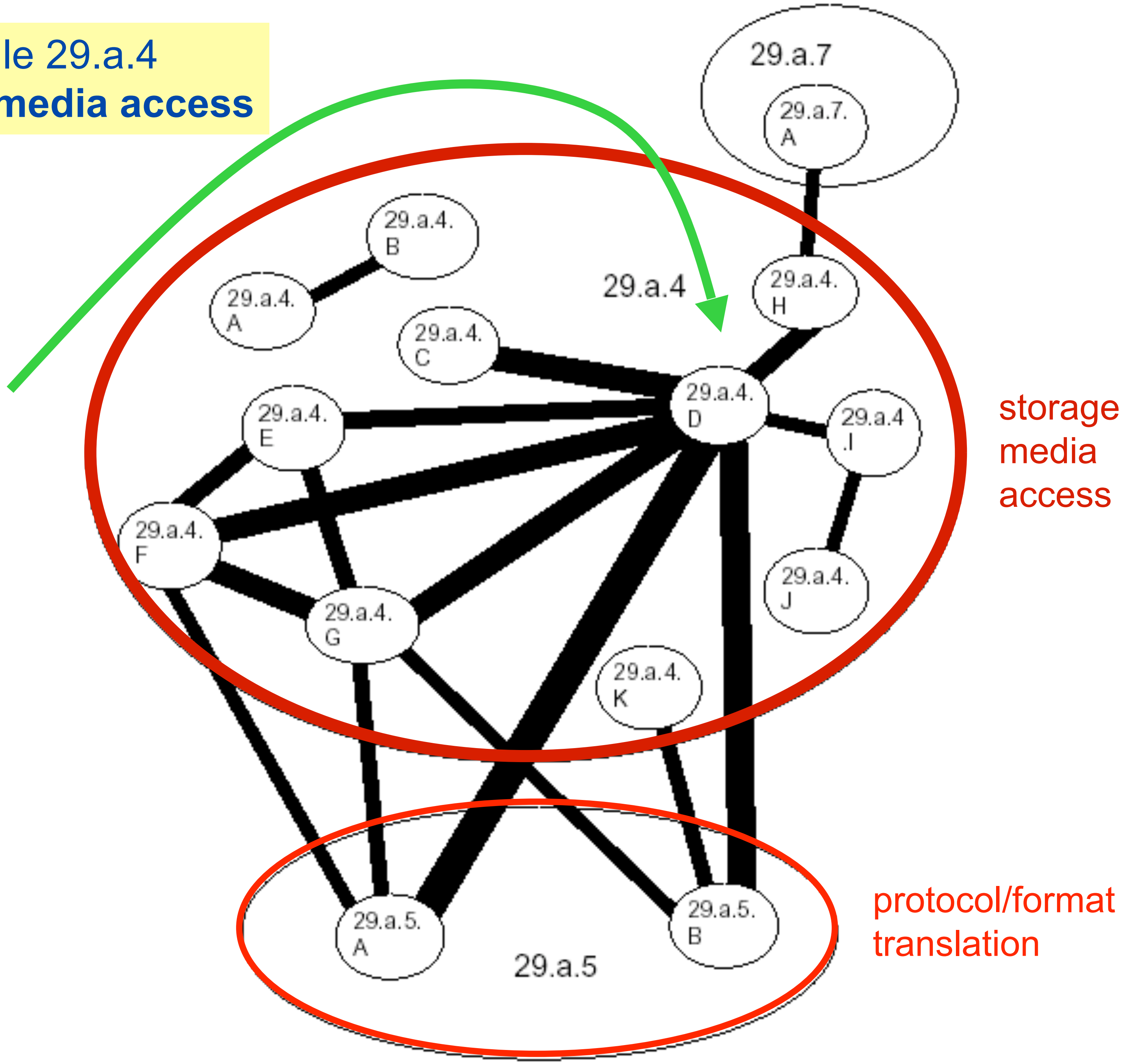
Submodule 13.c.9
GUI



Submodule 13.c.18
Printing framework

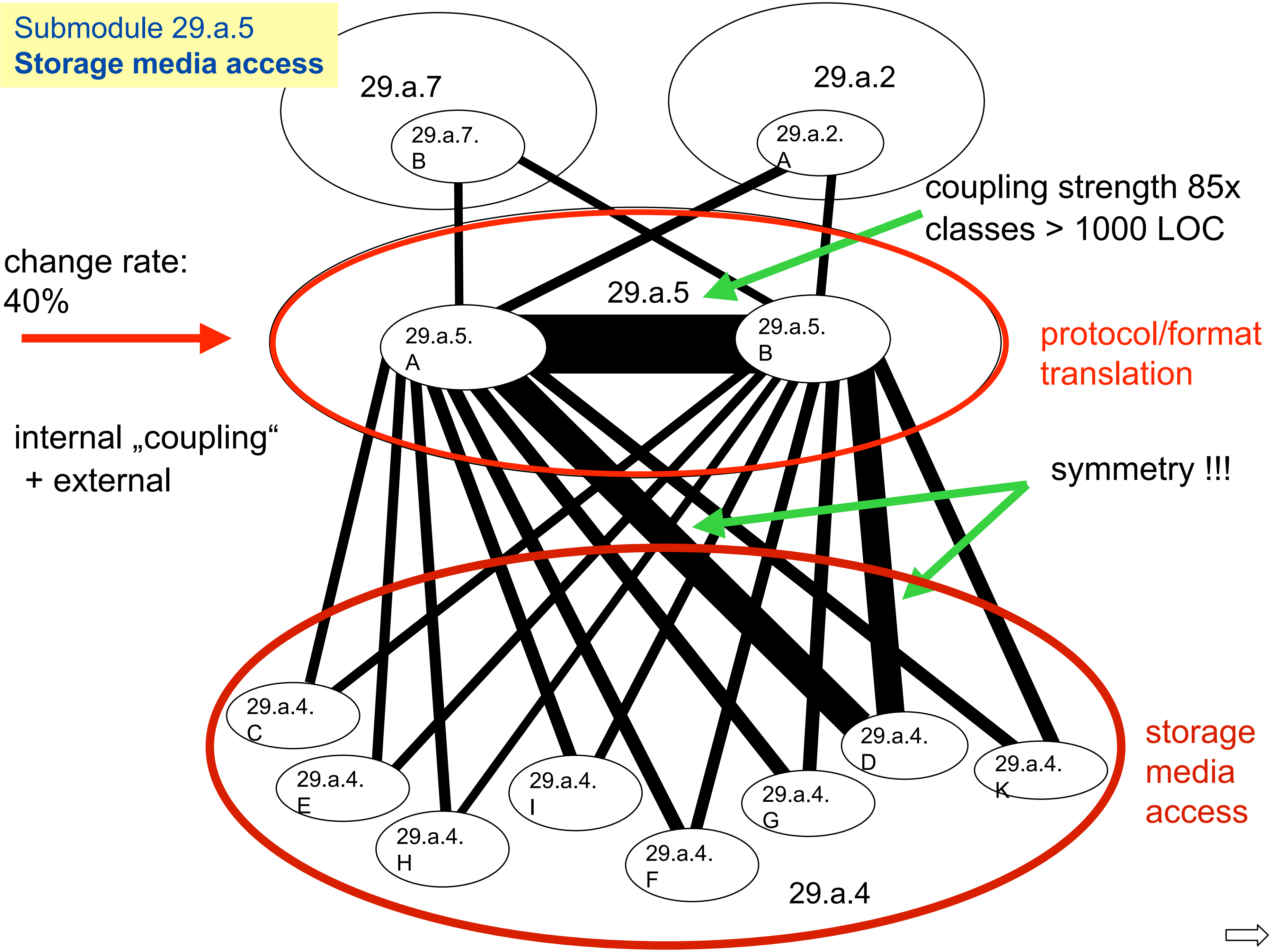


Submodule 29.a.4
Storage media access



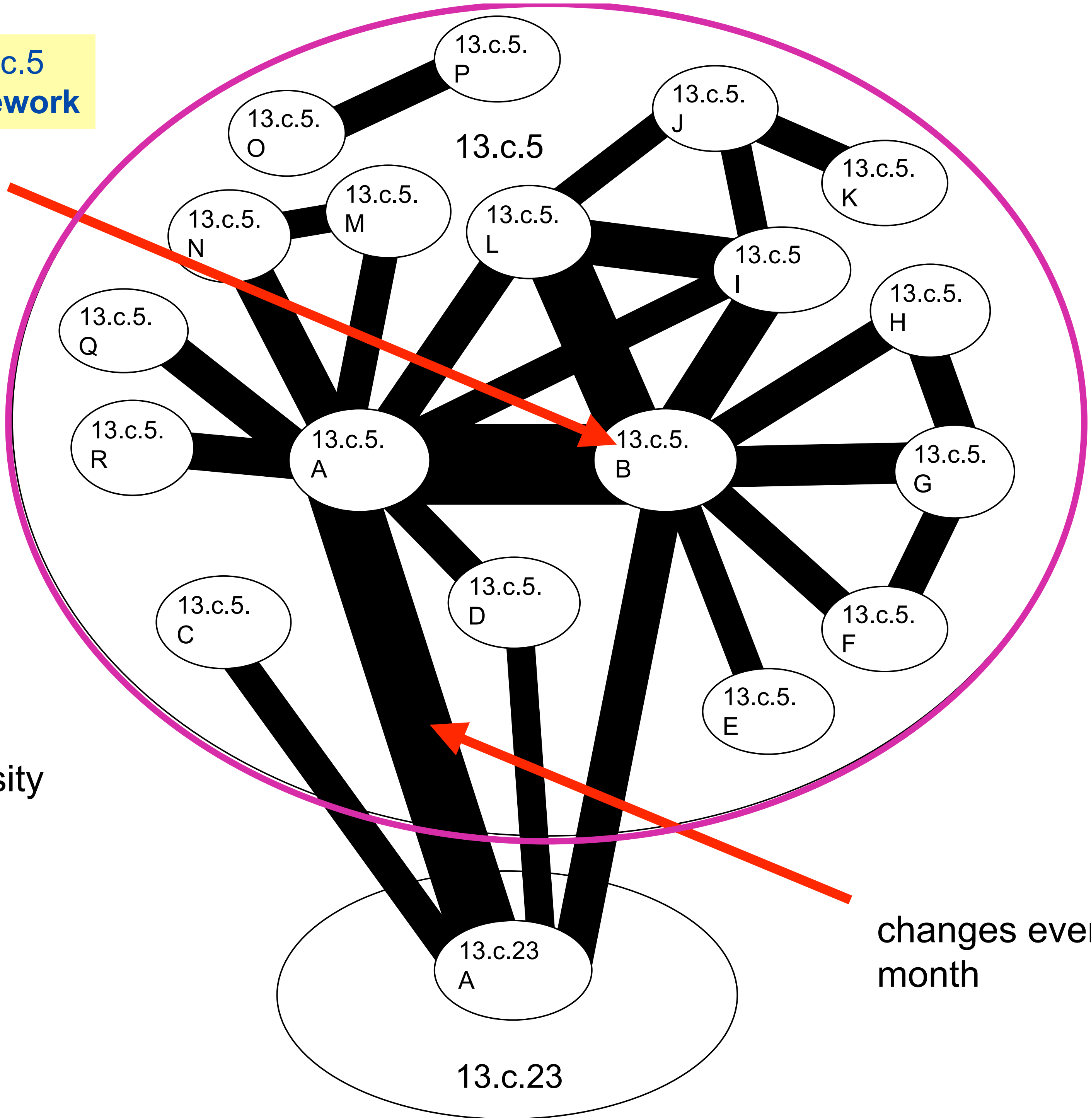
Submodule 29.a.5

Storage media access



Submodule 13.c.5
Imaging framework

inheritance hierarchy



change rate 25%

change intensity 80%

changes every month

Résumé

- RA combines all levels of decomposition: from classes to (sub-)modules through their CVS change data
- RA reveals many couplings (internal and external)
- Points to "bad smells":
  - spaghetti code, bad inheritance hierarchies, poorly designed interfaces, god classes, etc.
- Visualization simplifies understanding and navigation
- Domain knowledge an important ingredient for interpretation

Populating a Release History Database

Providing qualitative history data
for reasoning and visualization

Building a Release History DB

- 3 main sources:
  - Modification reports (MR) ← CVS
  - Problem reports (PR) ← Bugzilla
  - program and patch information ← release packages
- Relevant MRs and PRs are filtered, validated and stored in a Release History DB (RHDB)
- Problem: Identify change dependencies among system parts

Version control data

- A release represents a snapshot of the CVS repository for given software system
  - release number of product;
  - revision number of each file
  - for every release there is a symbolic name
  - branches as self-maintained lines of development

Case study: Mozilla

- Mozilla (www.mozilla.org)
  - analysis period: 1999-2002
  - 2.500 subdirectories
  - Source code: ~ 36.000 files, > 2 MLOC
  - CVS repository (revision information and modification reports)
  - > 180.000 bug reports
  - > 430.000 modification reports
  - Particular profiling data
  - Online documentation: roadmap, release notes, design documents

Example log-file from Mozilla source tree

```
RCS file: /cvsroot/mozilla/layout/html/style/src/nsCSSFrameConstructor.cpp,v
Working file: nsCSSFrameConstructor.cpp
head: 1.804
branch:
symbolic names:
   MOZILLA_1_3a_RELEASE: 1.800
   NETSCAPE_7_01_RTM_RELEASE: 1.727.2.17
   PHOENIX_0_5_RELEASE: 1.800
   ...
   RDF_19990305_BASE: 1.46
   RDF_19990305_BRANCH: 1.46.0.2
keyword substitution: kv
total revisions: 976; selected revisions: 976
description:
----------------------------
revision 1.804
date: 2002/12/13 20:13:16; author: doe@netscape.com; state: Exp; lines: +15 -47
Don't set NS_BLOCK_SPACE_MGR and NS_BLOCK_WRAP_SIZE on ...
----------------------------
...
----------------------------
revision 1.638
date: 2001/09/29 02:20:52; author: doe@netscape.com; state: Exp; lines: +14 -4
branches: 1.638.4;
bug 94341 keep a separate pseudo frame list for a new pseudo block or inline frame ...
----------------------------
....
```

Bugzilla bug reports

- bug id: This ID is referenced in modification report. Since the IDs are stored as free text in the CVS repository, the information can not be reliably recovered from the change report database.
- bug status (status whiteboard): Describes the current state of the bug and can be *unconfirmed*, *assigned*, *resolved*, etc.
- product: Determines the product which is affected by a bug. Examples in *Mozilla* are Browser, MailNews, NSPR, Phoenix, Chimera, etc.
- component: Determines which component is affected by a bug. Examples for components in *Mozilla* are Java, JavaScrip, Networking, Layout, etc.
- dependson: Declares which other bugs have to be fixed first, before this bug can be fixed.
- blocks: List of bugs which are blocked by this bug.
- bug severity: blocker, critical, major, minor, trivial, enhancement
- target milestone: Possible target version when changes should be merged into the main trunk.

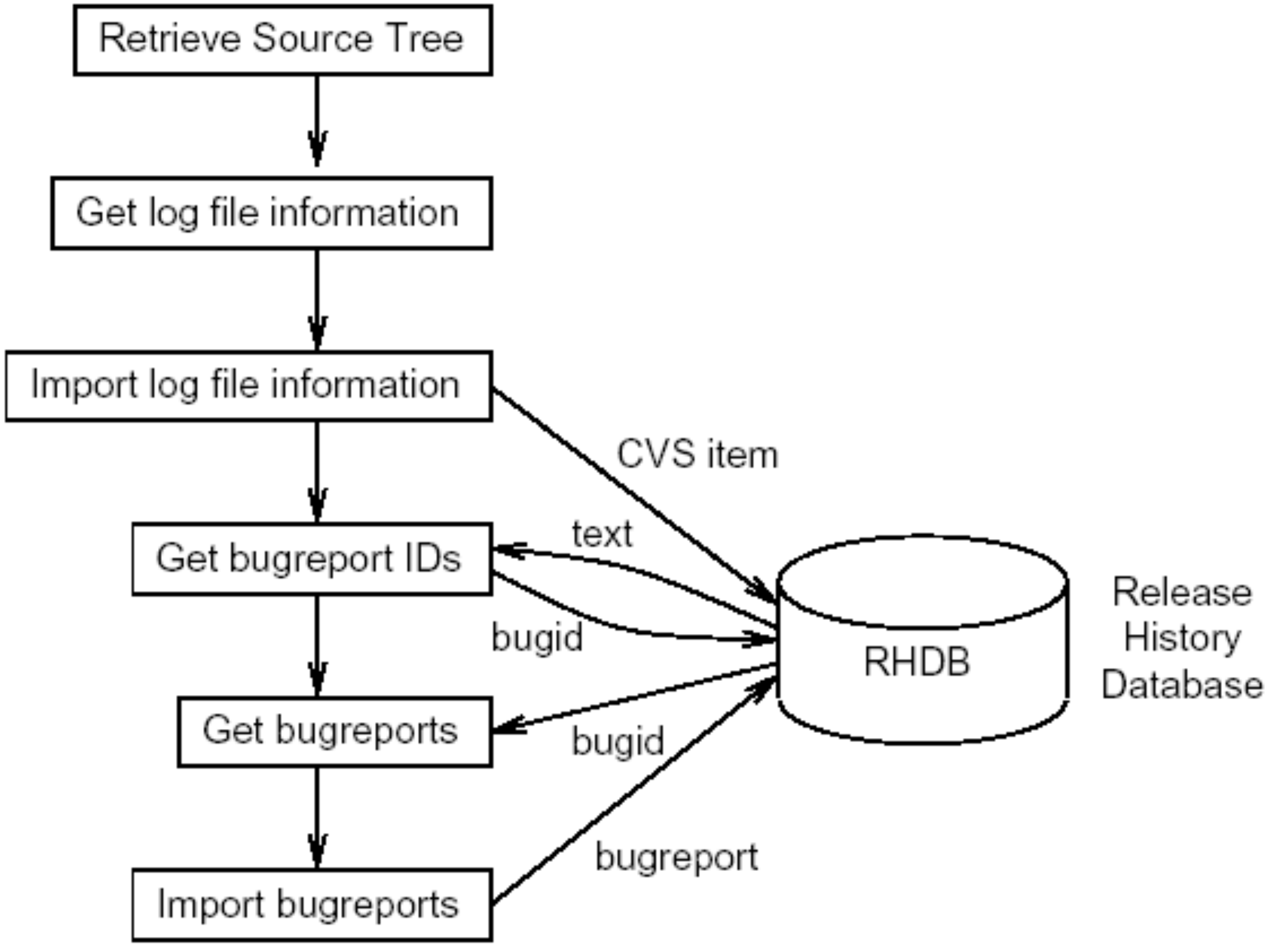
Bugzilla snippet

<bug_id> 100069
<bug_status> VERIFIED
<product> Browser
<version> other
<rep_platform> All
<assigned_to> doe@mozilla.org
<delta_ts> 20020116205154
<component> Printing: Xprint
<reporter> doe@mozilla.org
<target_milestone> mozilla0.9.6
<bug_severity> enhancement
<creation_ts> 2001-09-17 08:56
<qa_contact> doe@mozilla.org
<op_sys> Linux
<resolution> FIXED
<short_desc> Need infrastructure for new print dialog
<keywords> patch, review
<dependson> 106372
<blocks> 84947
<long_desc>
<who> doe@mozilla.org
<bug_when> 2001-09-17 08:56:29
</long_desc>

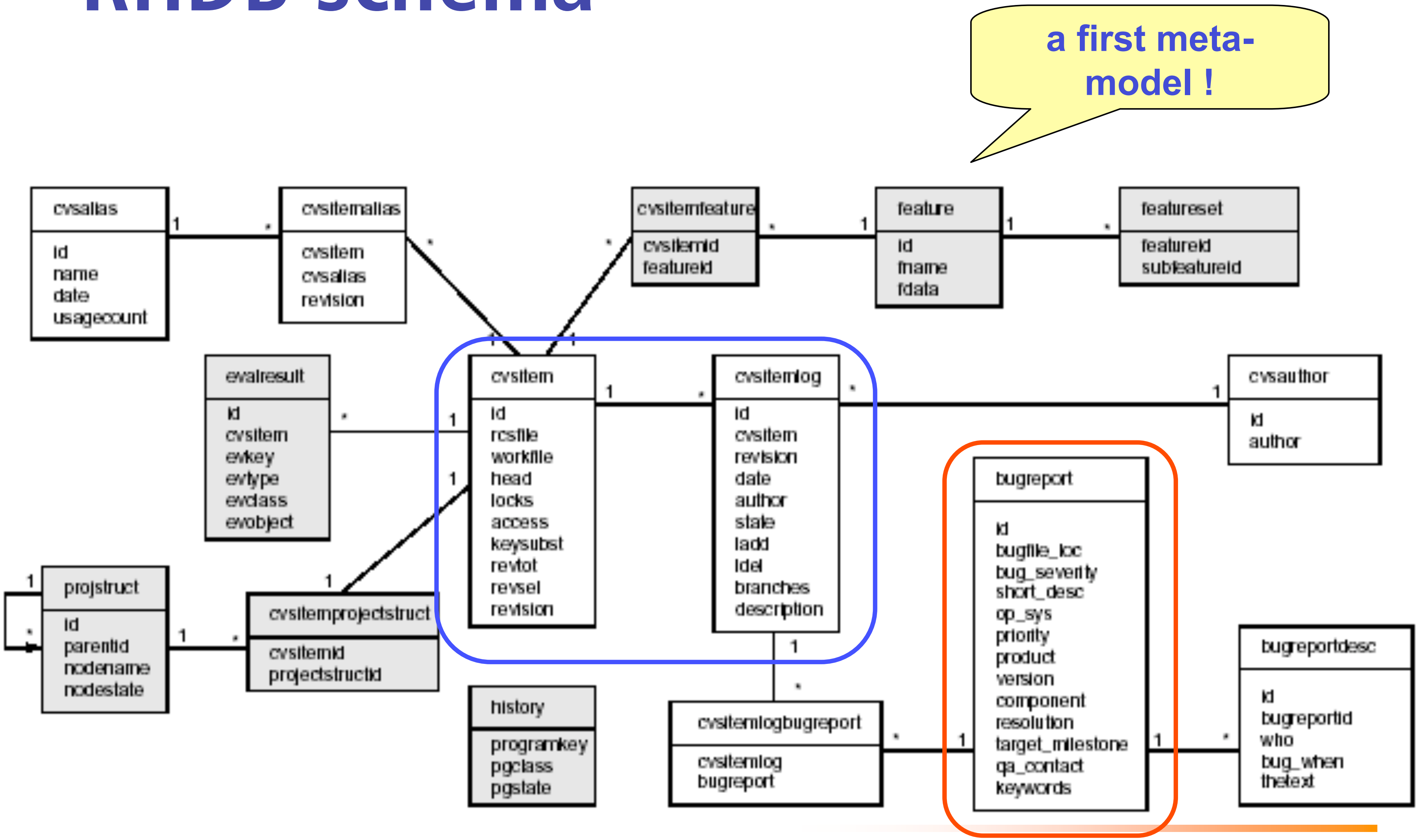
Populating a Release History DB

- Problem = re-establishment of links between modification reports (MRs) and problem reports (PRs) since no mechanisms provided by CVS
- We used the PR-IDs found in the MRs of CVS
- PR-IDs in MRs are detected using a set of regular expressions. A match is rated according to the confidence value: **high (h)**, **medium (m)**, or **low (l)**
  - confidence is considered high if expressions such as <keyword><ID> can be detected
  - confidence is considered low a six digit number just appearing somewhere in the text of a modification report without preceding keyword

Import process



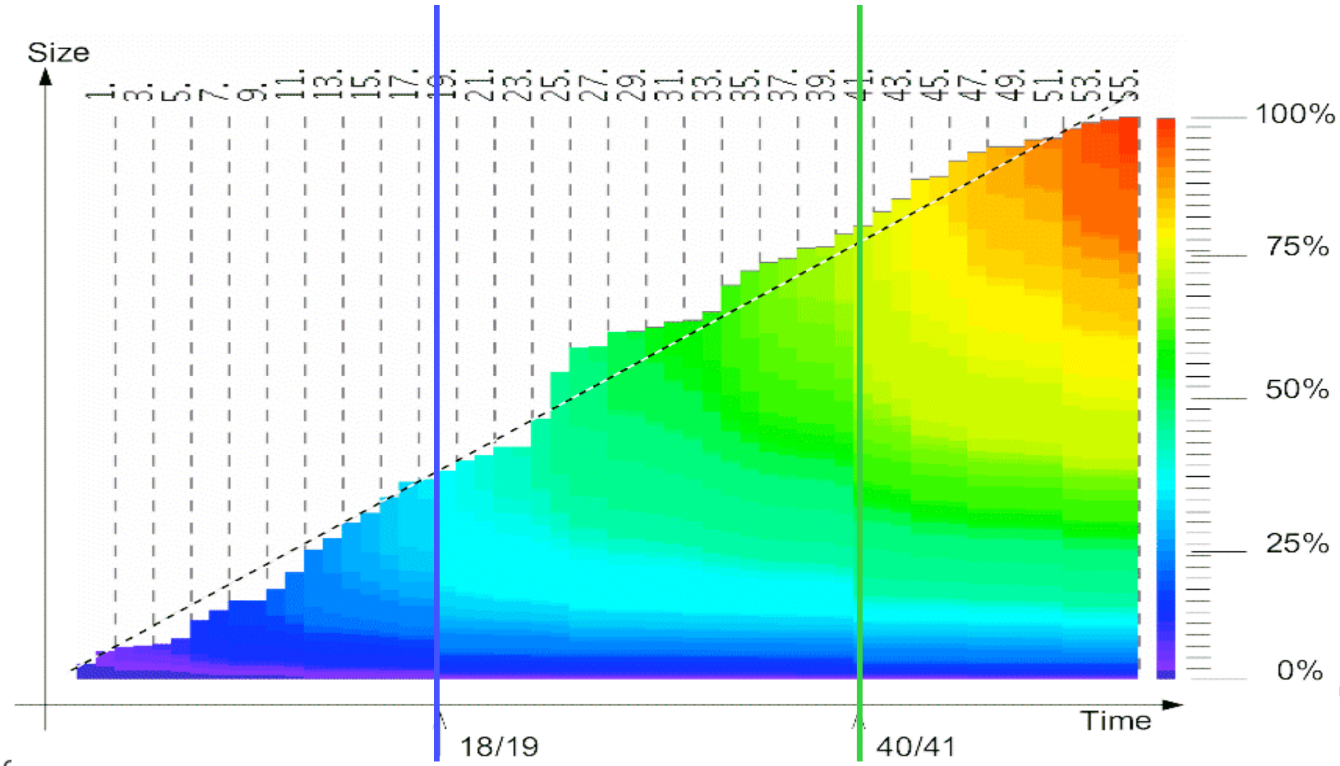
RHDB schema



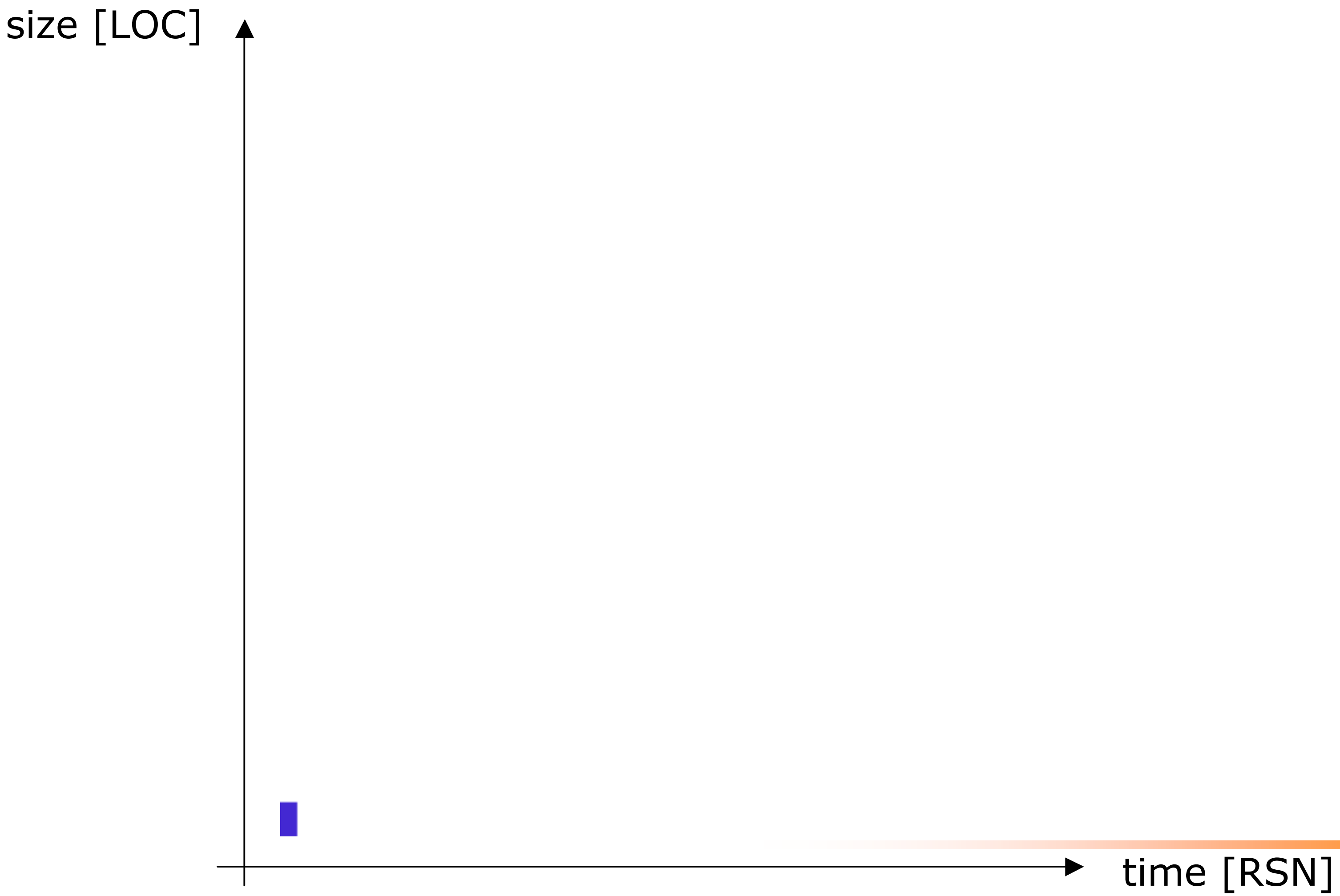
a first meta-model !
cvsalias
id
name
date
usagecount
cvsitemalias
cvsitem
cvsalias
revision
cvsitemfeature
cvsitemid
featureid
feature
id
fname
fdata
featureset
featureid
subfeatureid
evalresult
id
cvsitem
evkey
evtype
evclass
evobject
cvsitem
id
rcsfile
workfile
head
locks
access
keysubst
revtot
revsel
revision
cvsitemlog
id
cvsitem
revision
date
author
state
ladd
ldel
branches
description
cvsauthor
id
author
projstruct
id
parentid
nodename
nodestate
cvsitemprojectstruct
cvsitemid
projectstructid
history
programkey
pgclass
pgstate
cvsitemlogbugreport
cvsitemlog
bugreport
bugreport
id
bugfile_loc
bug_severity
short_desc
op_sys
priority
product
version
component
resolution
target_milestone
qa_contact
keywords
bugreportdesc
id
bugreportid
who
bug_when
thetext

Views on Mozilla evolution

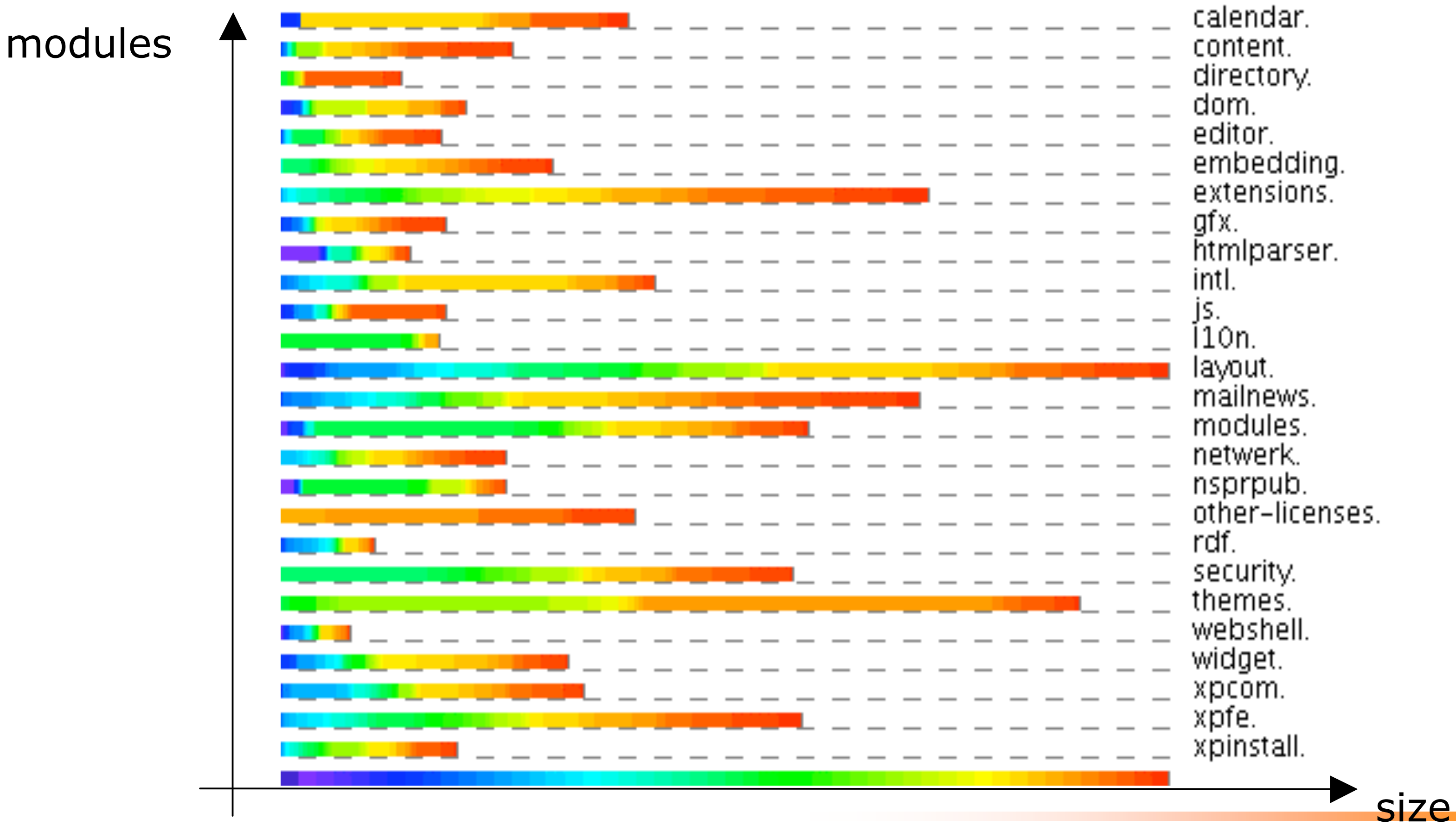
- 50% of files have been modified in last quarter of observation
- although only 25% of files have been integrated



Mozilla evolution



Views on Mozilla evolution /2



Résumé

- RHDB offers some characteristics for evolution analysis
  - linkage between changes and bugs (qualified)
  - files logically coupled via changes and bugs
  - branch/merge revision data

- data set as a basis for further analyses and visualizations (e.g. MDS-view)

- a basis for data exchange among research groups in the direction of a meta-model for release data

Multi-dimensional Visualization of Evolution Data

Allowing to locate hidden feature and module dependencies

Selection of Problem Reports

- Filtering those concerned with admin issues
  - "license foo" (PR-ID #98089, 7961 referenced files)
  - "printfs and console window info needs to be boiled away for release builds" (#47207, 1135), or
  - "Clean up SDK includes" (#166917, 888)
  - "repackage resources into jar files" (#18433, 289)
- We used 255 as limit for the amount of bug reports to be accepted
  - no major or critical PRs filtered

Feature evolution

- Goal of the feature extraction process is to map the abstract concept of features onto a concrete set of files which implement a certain feature.
  - *… an observable and relatively closed behavior or characteristic of a (software) part* [16]
- We first created a single statically linked version of *Mozilla* (*v1.3a* with the official freeze date 2002-12-10)
  - with profiling support enabled. From several test-runs where the defined scenarios were executed, we created the call graph information using the GNU profiler.
- The call graph information again was used to retrieve all functions and methods visited during the execution of a single scenario.

Scenarios and features

Scenario	Description	Feature	Color	Files
Core	mozilla start / blank window / stop	Core	White	705
HTTP	TrustCenter.de via HTTP[1]	Http	DeepPink	28
HTTPS	TrusterCenter.de via SSL/HTTP[2]	Https	MediumGreen	6
File	read TrustCenter.de from file	-	-	-
MathML	mathematic in Web pages[3]	MathMlExtension	YellowGreen	13
About	"about:" protocol	About	Gold	3
PNG	sample image[4]	ImagePNG	DarkOrange	10
XML	XML Base[5]	Xml	MediumOrchid	65
JPG	JPEG Karlskirche[6]	ImageJPG	Cyan	16
fBlank	read blank html page from file[7]	Html	DeepSkyBlue	76
hBlank	blank html page via HTTP[8]	-	-	-
ChromeGIF	Mozilla logo[9]	ImageGIF	SlateBlue1	4
Image	-	Image	OrangeRed1	3

Multidimensional scaling

- the goal is to map objects to points in such a way that given dissimilarities are well approximated by the distances
  $|| x_i – x_j ||$ in a k-dimensional solution space.

- minimization of a stress function

- A problem report descriptor d_i of a problem report p_i is built of all artifacts a_n which refer to a particular problem report via their modification reports m_k
  - $d_i = \{a_n \mid a_n \; R \; m_k \wedge m_k \; R \; p_i \}$

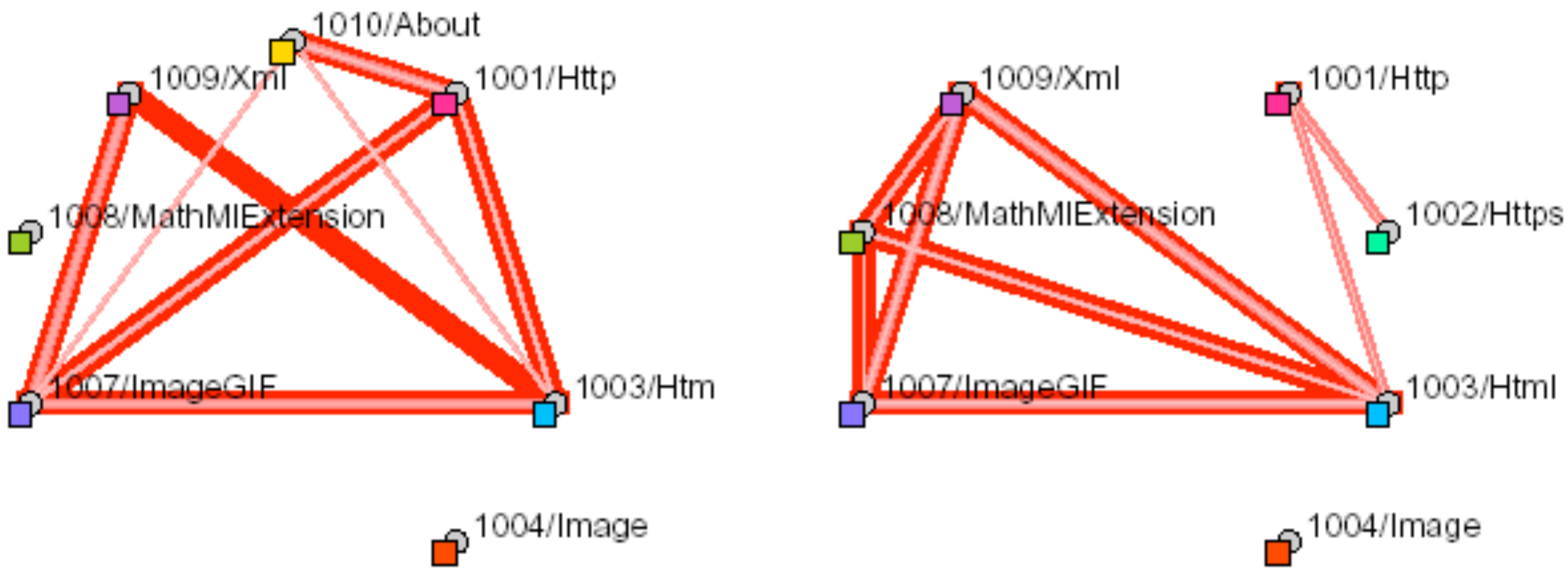
- distance data for every pair of problem report descriptor $<d_i, d_j>$ are computed

- XGvis: a system for multidimensional scaling and graph layout in any dimension (research.att.com)

Views

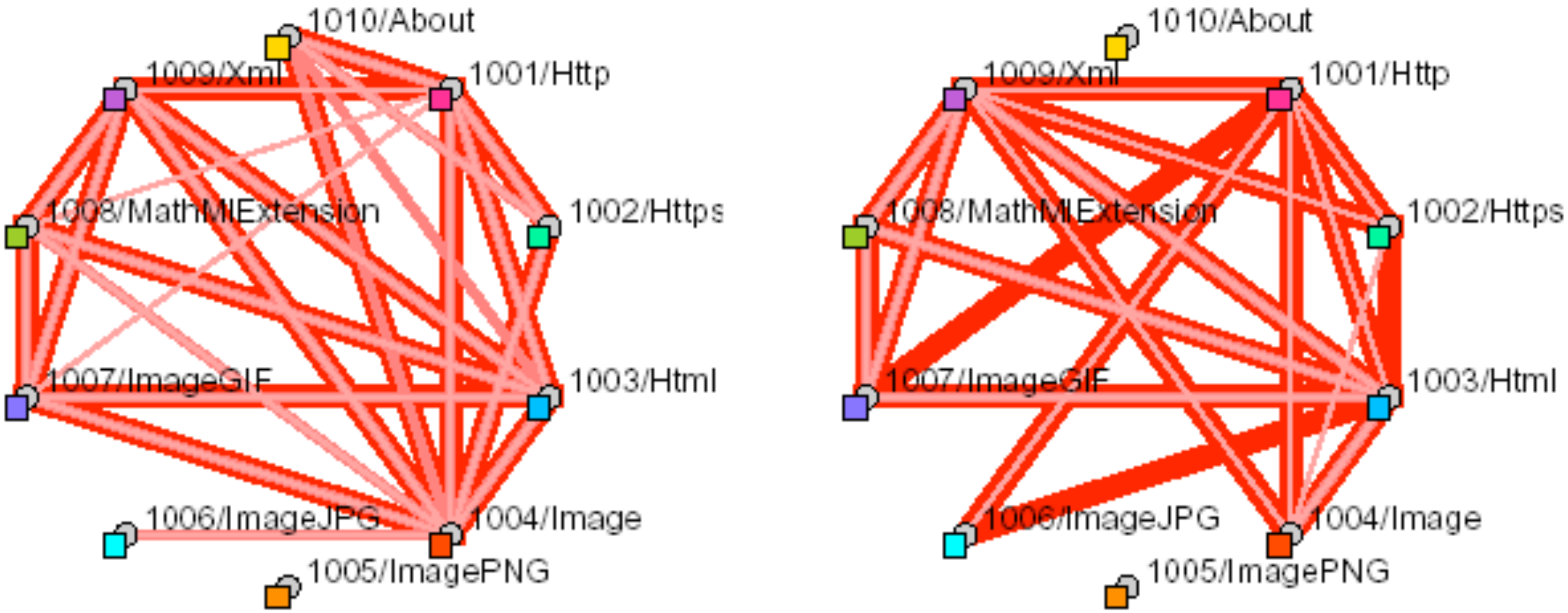
- *feature-view* focuses on the problem report based coupling between the selected features
- *project-view* depicts the reflection of problem reports onto the structure of the project-tree (ie. directory structure)

Feature View /1



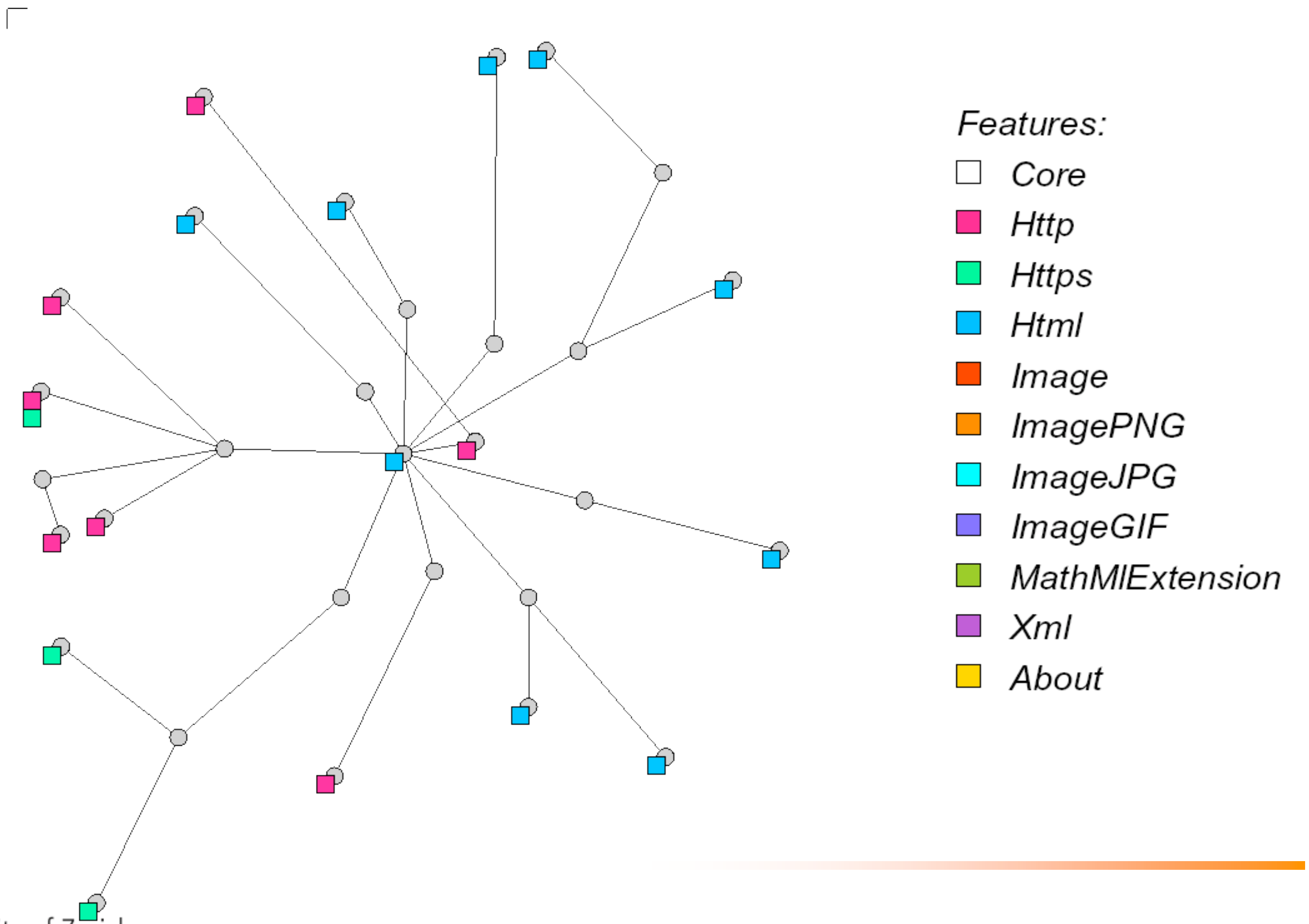
(a) 1999 (b) 2000

Feature View /2

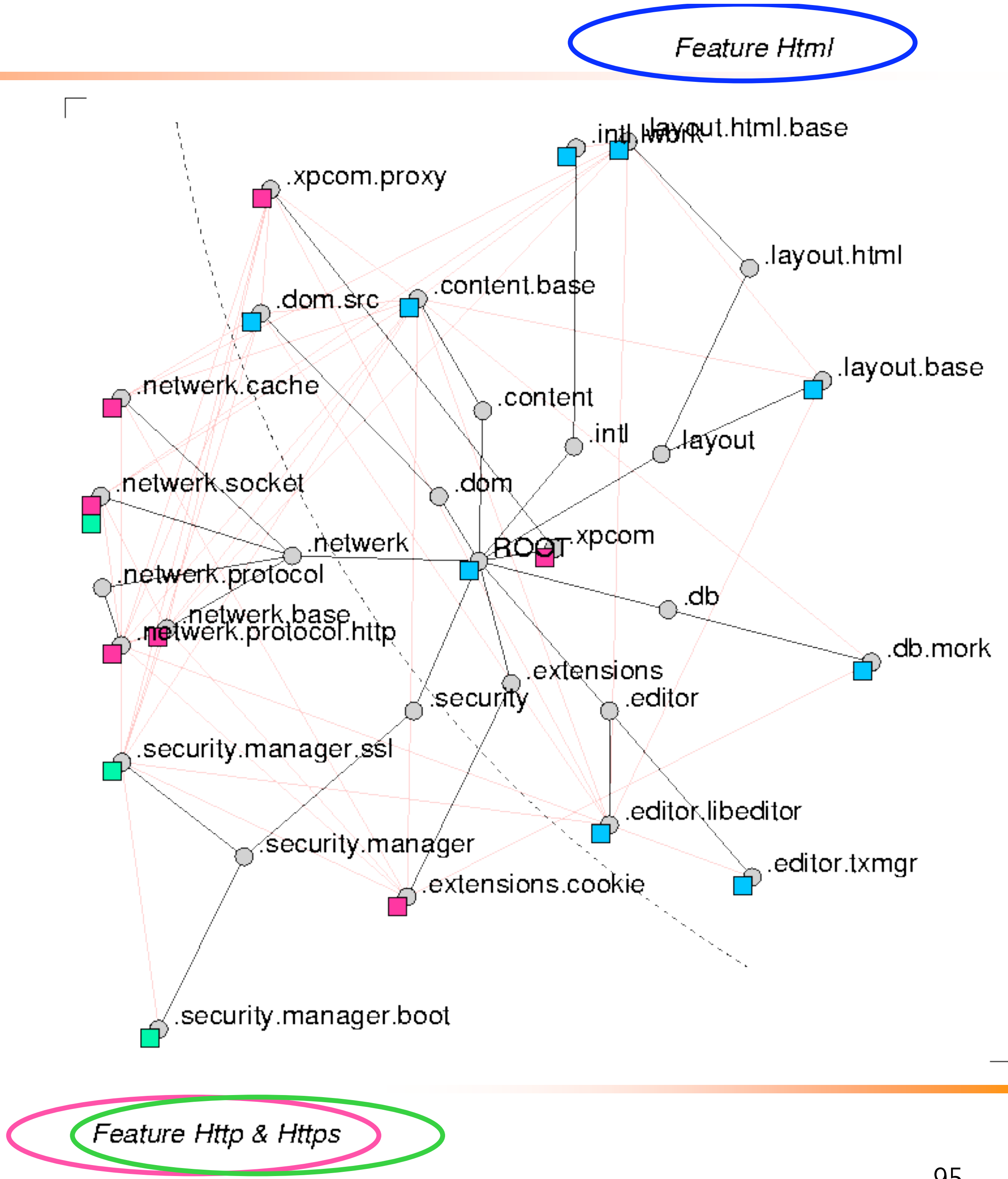


(a) 2001 (b) 2002

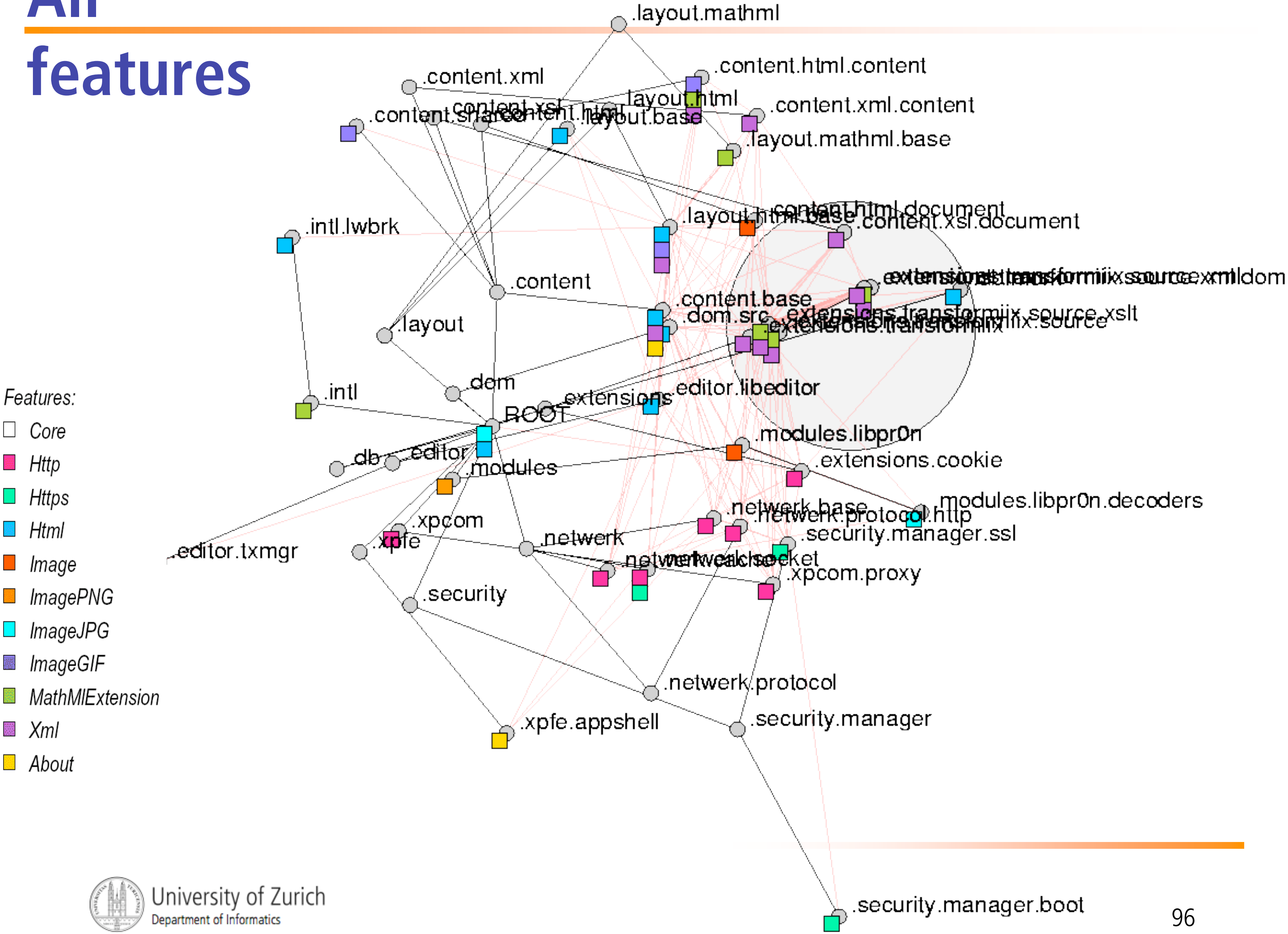
Project view – structure & features



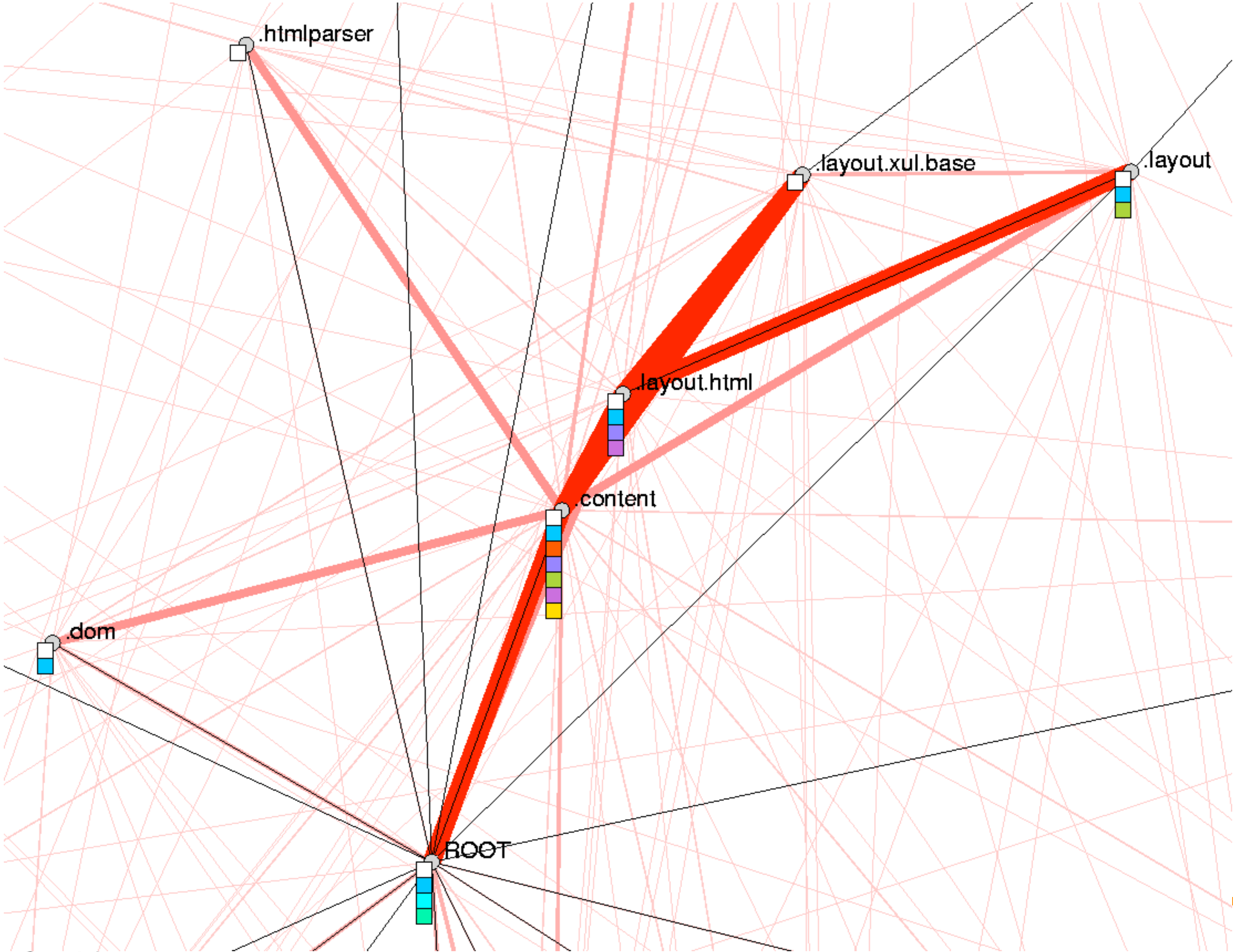
Http, https, html



All features



core & features



Results

- Final visualization:
  - all problem reports rated major or critical;
  - number of PR references: >= 50
  - resulting graph: 25 nodes, 215 edges via PRs
- Most critical subsystems are concerned with visualization – that's what we have seen via MDS
- Nodes with highest density in severe PRs are
  - .content (595 references)
  - .layout.html (438); .layout.xul.base (220); .layout (210)
- Spreading of edges

Conclusions

- Software Evolution Analysis
  - integrates quantitative analysis and common change sequence analysis
  - helps to identify different types of architectural shortcomings
  - in combination with graphical representation facilitates the understanding of certain syste, characteristics
  - allows reasoning about a software system on a macro level (no source code analysis)
  - requires very little data to be kept

- Integrating modification reports and bug reports with feature profiling reveals many couplings
  - couplings among features (feature view)

Next steps

- Refine analysis and enhance visualization and navigation
- Integrate with other evolution analyses and evolution data in a SEA framework
  - bug report data
  - modification report data
  - test data and properties
  - feature information
  - multi-dimensional visualization

- Additional case studies
- More Papers at seal.ifi.unizh.ch/publications/

References

- [Godfrey 2001]