Software Bugs and Evolution: A Visual Approach to Uncover Their Relationship

Marco D'Ambros and Michele Lanza

- Faculty of Informatics -
University of Lugano
Switzerland

CSMR 2006 - 10th European Conference on Software Maintenance and Reengineering, Bari, March 22-24, 2006

Introduction

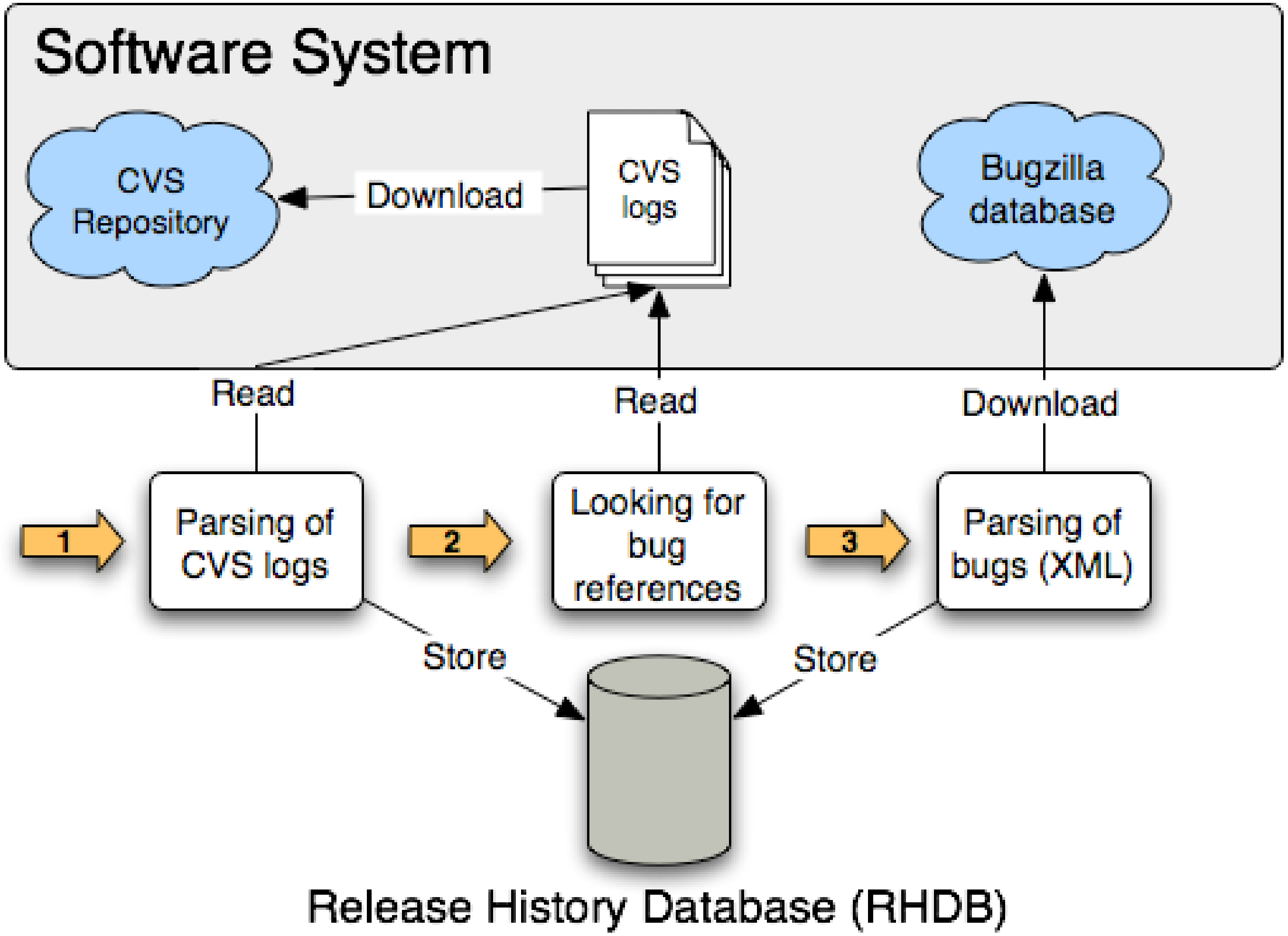
- Understanding the evolution of **large** *system* is a key issue in software industry
- The evolution is strongly coupled with the structure of the system

- The history of a system is described by several kinds of information, among which:
  - Source code history as recorded by CVS
  - Problem report as stored in Bugzilla

Goal

- Study the relationship between the evolution of the **source code** and the evolution of the **problem reports** at any granularity level
  - **Course grained:** To characterize system modules and compare them
  - **Fine grained:** To detect entities which revealed problems
  - **All levels:** To detect common patterns in the evolution of system entities

Data Retrieval

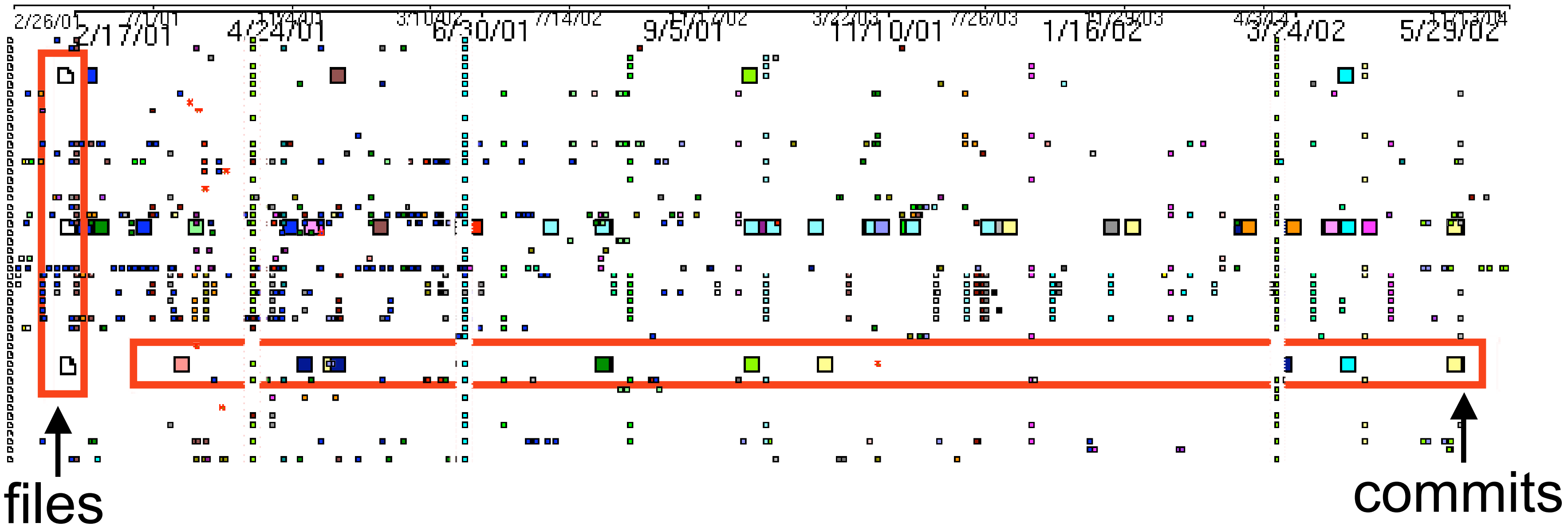


Dealing with Huge Amount of Data

- The RHDB for large systems can contain:
  - More than **15k** files information
  - More than **250k** commit-related information
  - More than **20k** problem report information

- We need an approach to deal with this amount of data
- We use a visualization technique which provides a lot of information in a condensed way

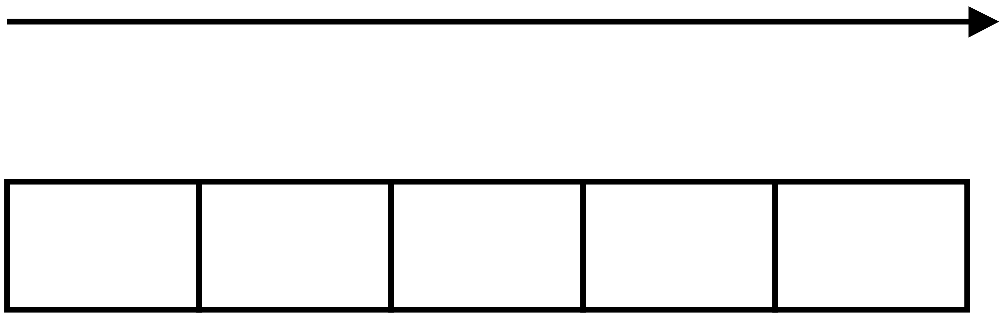
TimeLine View



Shortcomings:

- How to relate commit- and bug-related information?
- Scalability
- Aggregation

Time Discretization

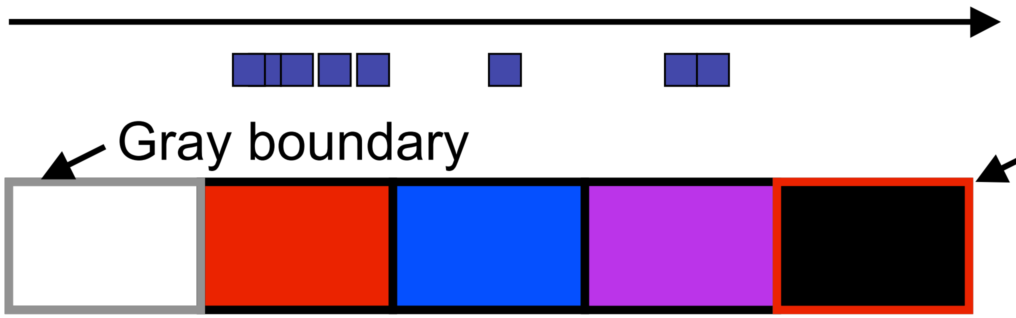


Continuous time

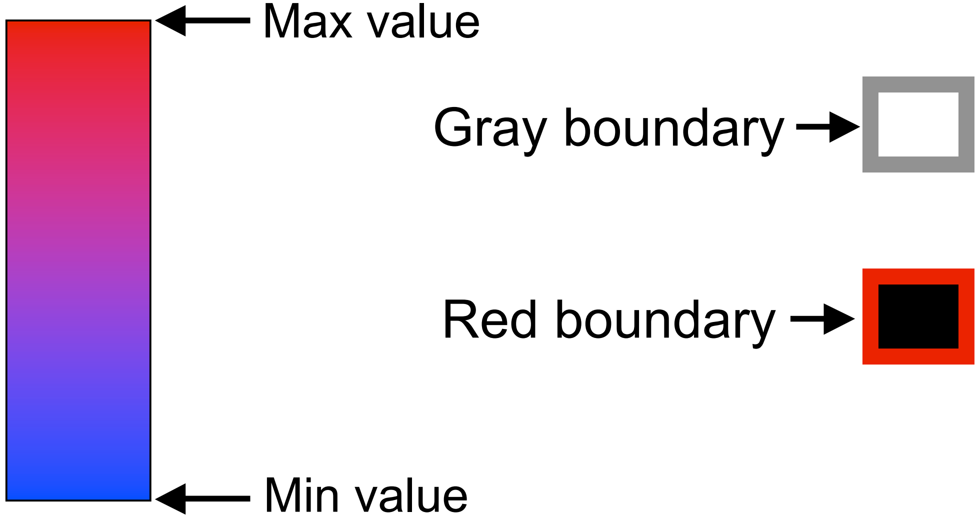
Discrete time

- The time is divided in intervals, represented by the rectangles
- All the intervals have the same size
- The size is parametrizable

Commits/Bug Reports Information



Color temperature mapping



Before first / after last commit / Bug report

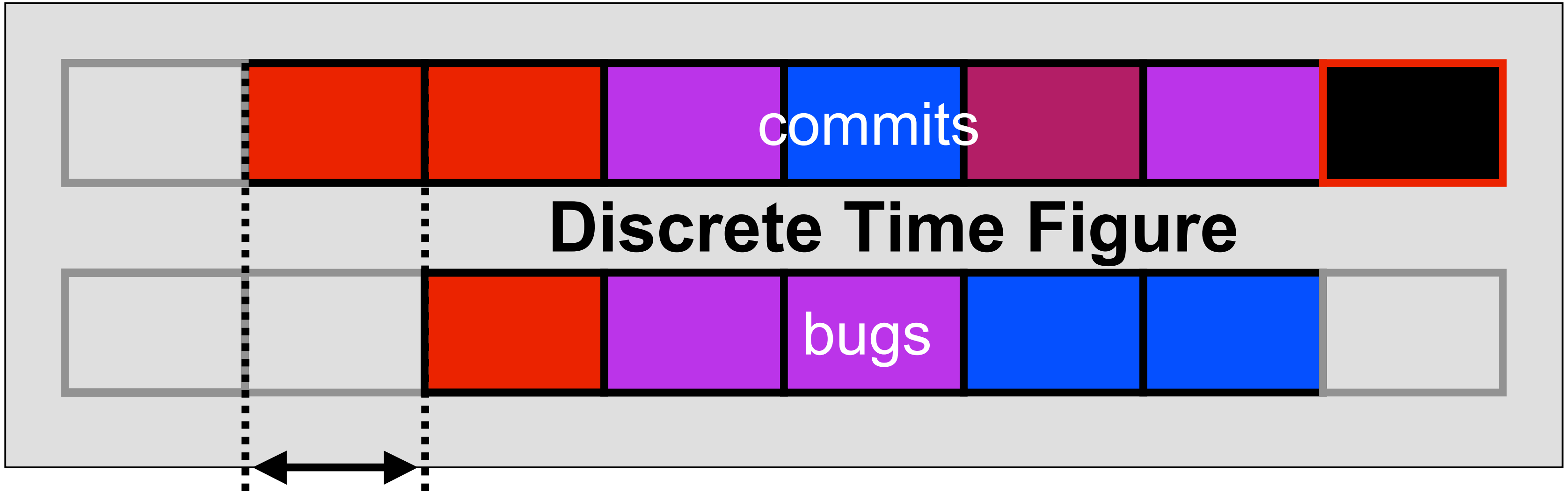
After removed from the system (only commit)

Combining the Information

We have seen how to show commit- and bug-related information

We want to combine the data to get a better understanding of the evolution

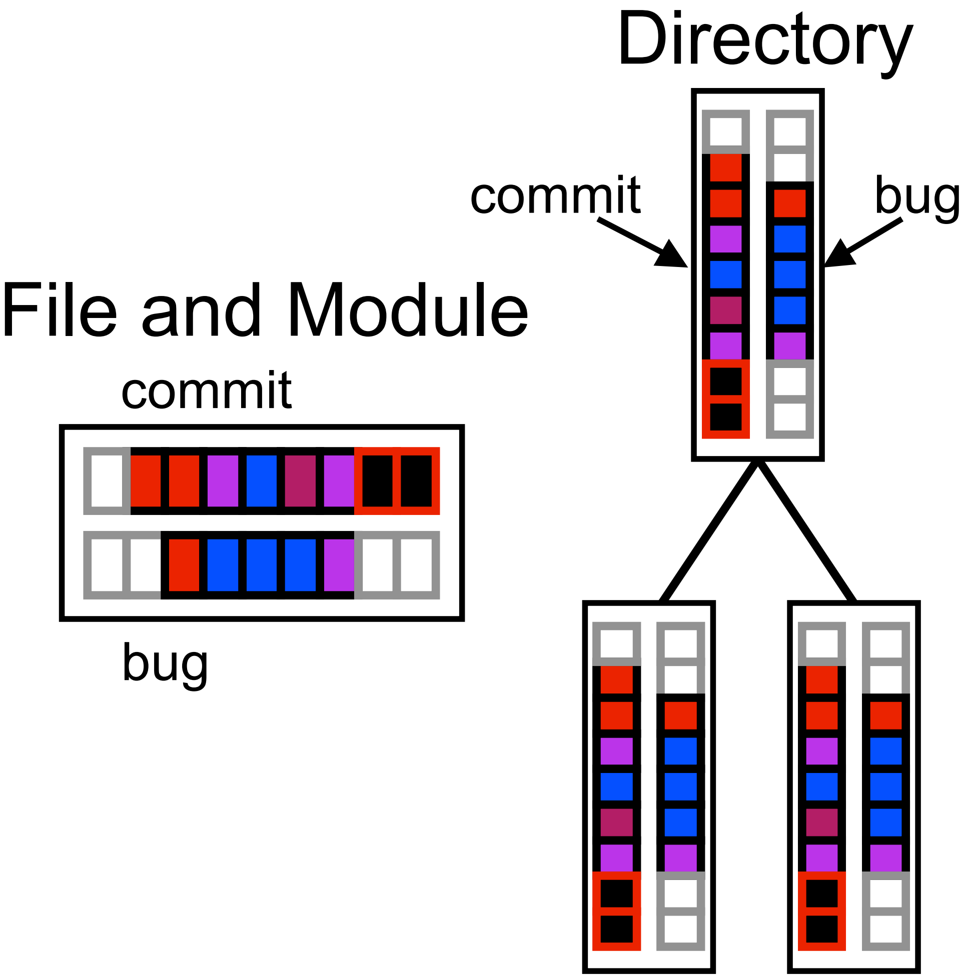
One **simple integrated** figure



Same time interval

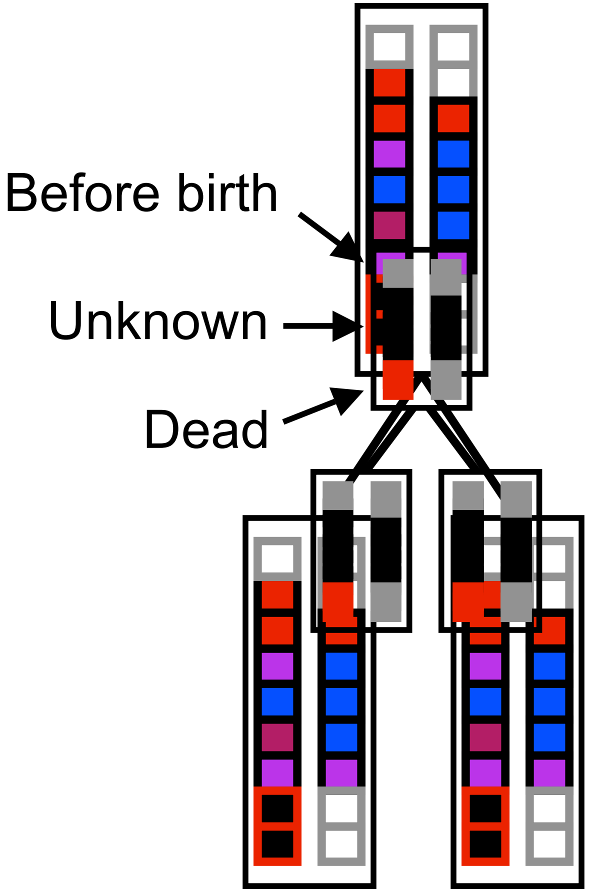
Dealing with any granularity levels

- The Discrete Time Figure is applicable to any CVS artifacts (file, directory, module)
- For a directory or a module we count the number of commits (bug reports) for all the files belonging to it

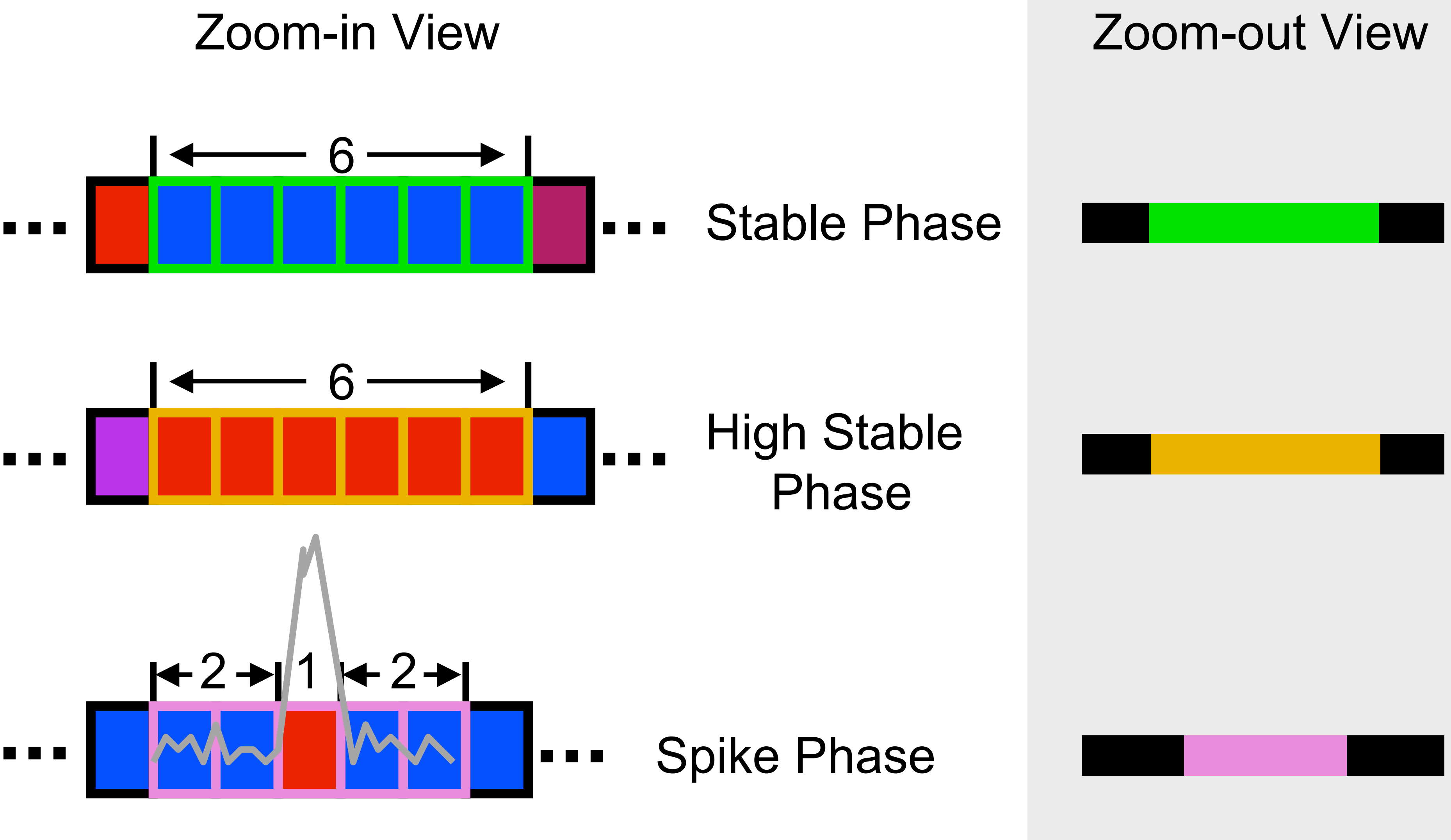


Scalability Issues

- For visualizing many entities we need to **zoom-out**, but
  - The inner colors are lost, only the boundaries are visible
  - Only birth and death information are visible
- We need to introduce another level of abstraction



Discrete Time Figure Phases



Zoom-in View

Zoom-out View

Stable Phase

High Stable Phase

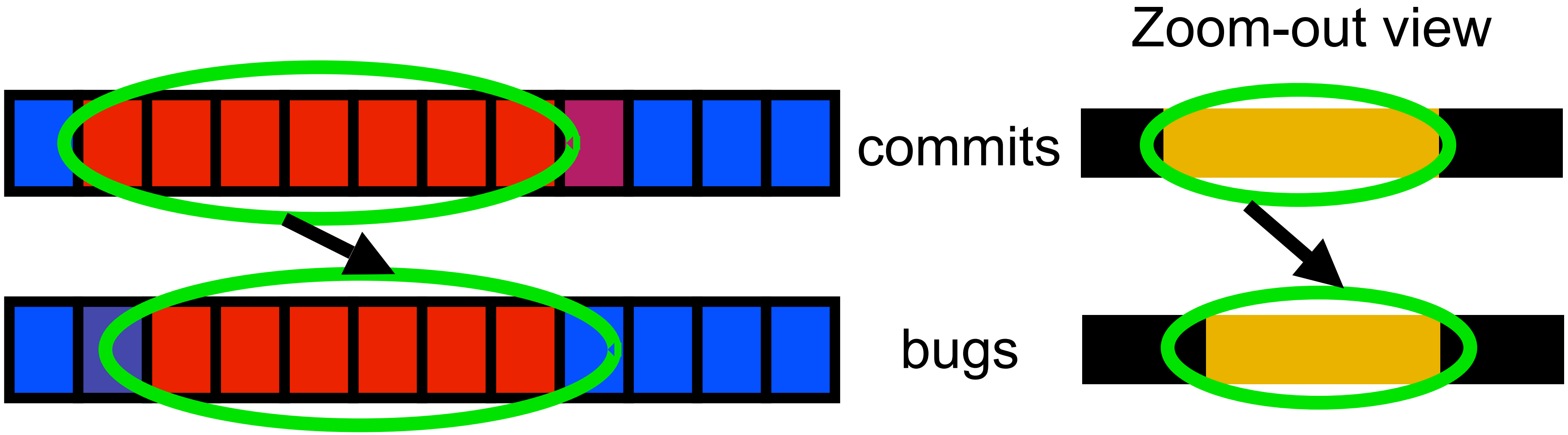
Spike Phase

Pattern Language

- Based on the Discrete Time Figure we define a pattern language
  - The patterns allows us to characterize the evolution of software entities
  - The patterns are based on the combination of commit- and bug-related information
  - 7 patterns are (*formally*) defined and presented in the paper
  - The patterns can be automatically detected by means of a query engine

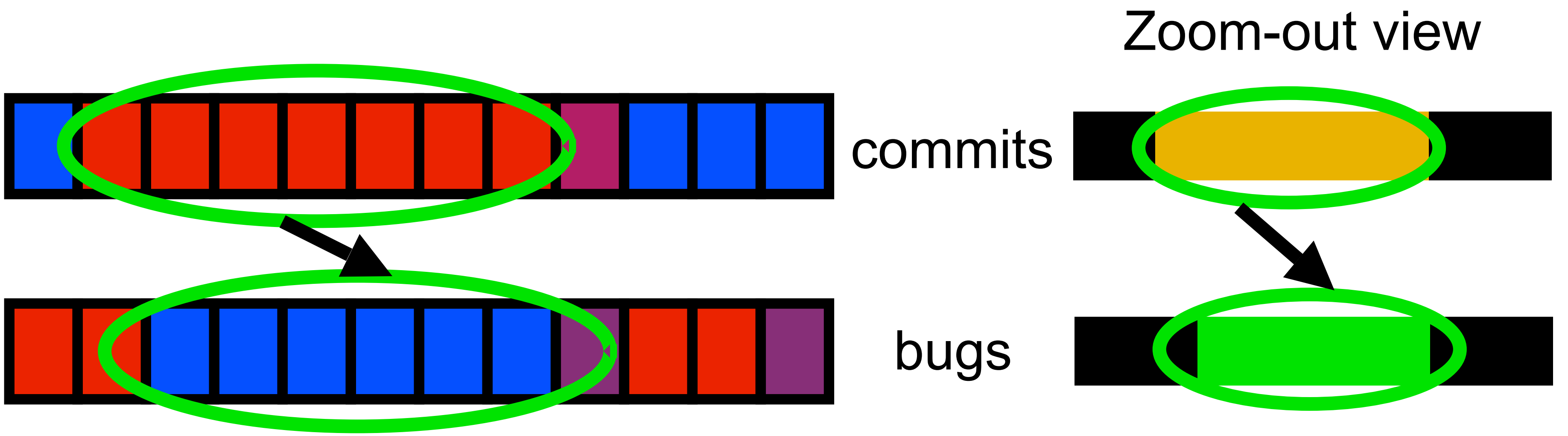
Addition of Features Pattern

- **Idea**: Introducing new features in the system is likely to introduce new bugs
- **Appearance**: an high stable phase in the commits (lots of commits) is followed by an high stable phase in the bug (lots of bugs)



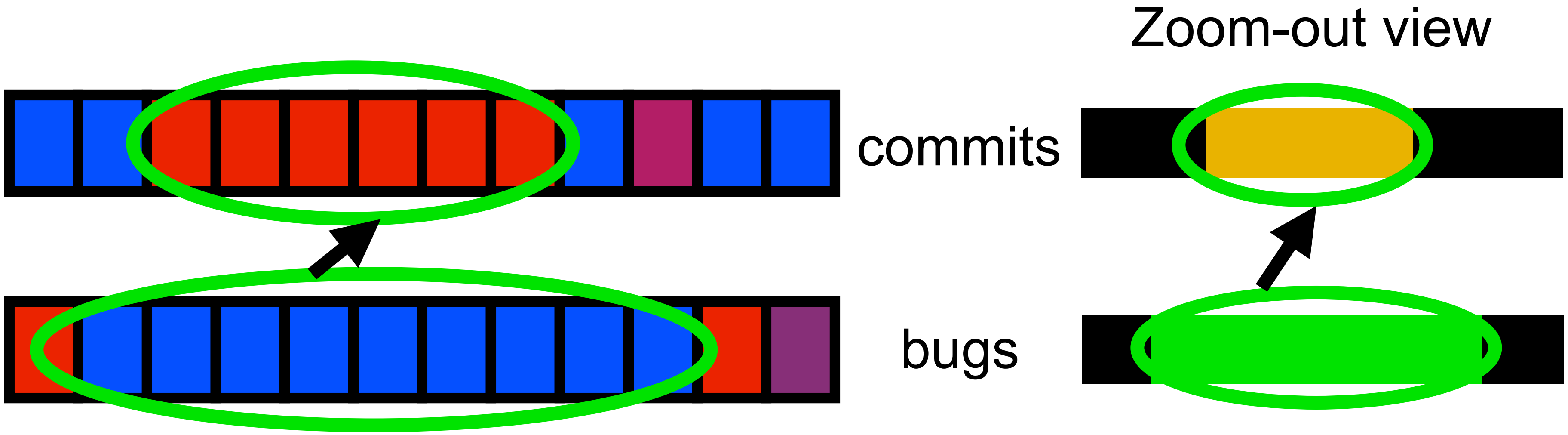
Bug Fixing Pattern

- **Idea**: The effort revealed by the increasing number of commits was spent to fix bugs
- **Appearance**: an high stable phase (lots of commits) in the commits is followed by a stable phase in the bug (few bugs)

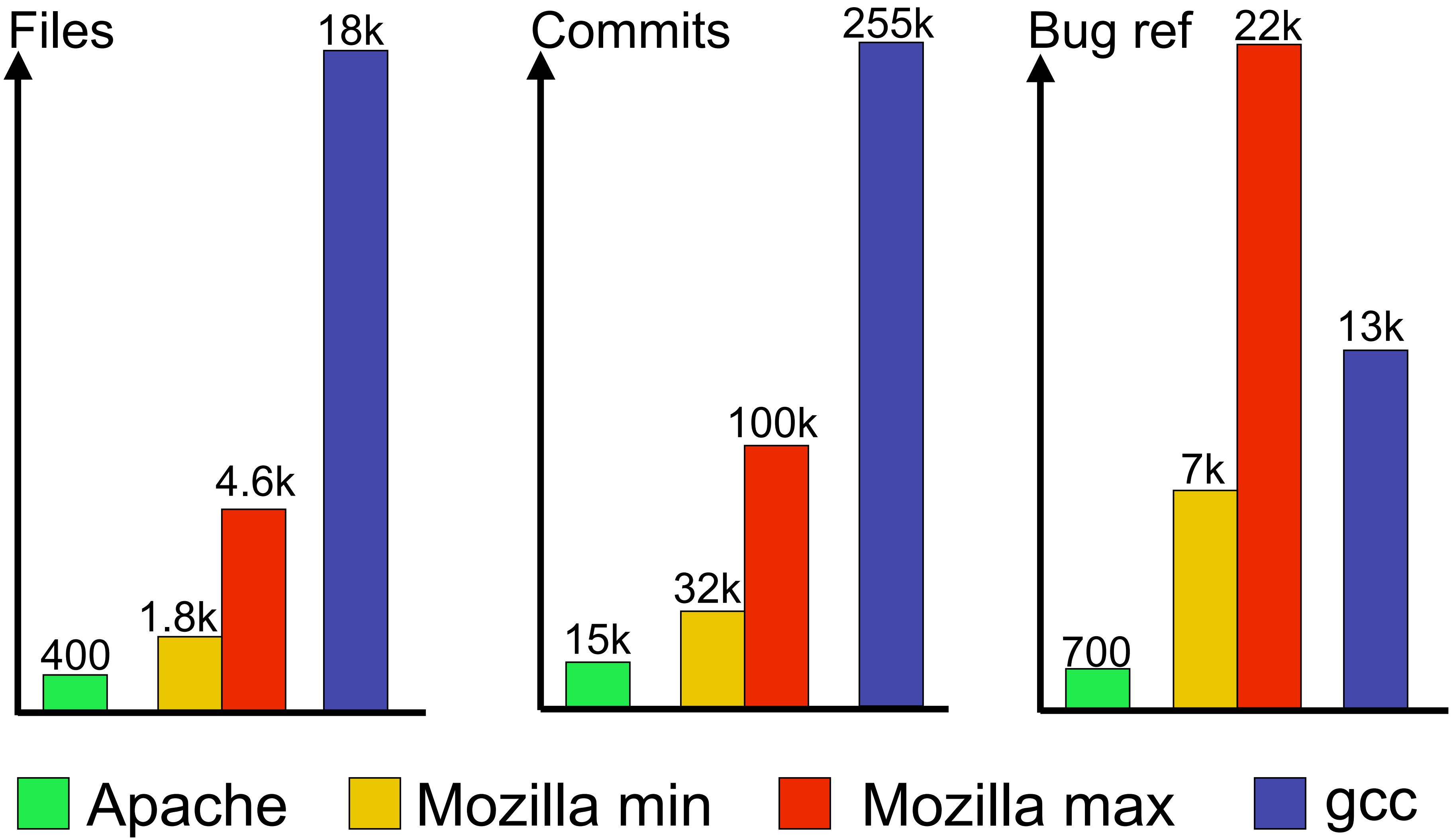


Refactoring/Code Cleaning Pattern

- **Idea**: Refactoring and code cleaning require an effort in terms of commits while they should not introduce new bugs
- **Appearance**: an high stable phase in the commits (lots of commits) is “contained” in a stable phase in the bugs (the number of bugs remain low)



Case Studies: Systems



Case Studies: Methodology

1. Build a view with all the directories of the target system → Granularity level: Directory
2. Apply a query engine on the view to detect all the defined patterns → Characterize the system in terms of patters
3. Analyze the view to understand how and where the patterns are distributed → Identify areas or entities for further inspection

Some Results

gcc

- **913** patterns on 1145 directories
- Fast changing
  - Lots of spike solutions
  - Few entities "survived" to all the changes

Mozilla (all 4 modules)

- **1586** patterns on 1647 directories
- 3 patterns appear with the same frequency
- *SeaMonkeyCore* has the max number of most of the patterns

•*Addition of features* is much more frequent than *Bug fixing* and *refactoring/code cleaning*

Pros and cons

+ Language independent approach
+ Automatic detection of phases
+ Scalability

- Hypotheses need to be verified
- The views can be difficult to read for inexperienced users

Conclusion

- The Discrete Time Figure technique
  - shows the relationship between source code and bugs evolution
  - indicates evolutionary patterns which can be automatically detected

- Future Work
  - Applying the technique on known case studies to get feedback
  - Decrease the granularity to the method/function level