Motivation

• Given a software system, continuously updated.

• How to keep all branches in sync, including releases deployed at the customer?
Approach 1: write patch for one release, apply elsewhere
Approach 2: use modules

Customer

```
1.3 | 1.2 | 1.2
----|-----|-----
1.2 | 1.1 | 1.2
```

Latest

```
1.3 | 1.2 | 1.2
----|-----|-----
1.2 | 1.1 | 1.2
```
Motivation (2)

• When upgrading a module, dependencies should still be satisfied.

• Fewer dependencies => easier to upgrade.

• Given a software system, are modules sufficiently independent such that they can be upgraded easily?
Upgrade dependencies

Module A

Module B

2012
today

New symbol added and used

Symbol removed
Example of upgrade dependencies

Two alternatives for B providing symbol S.

Objective: Choose a **minimal** set of upgrade dependencies (limit impact).
Approach outline

1. Gather syntactical interface usage data.

2. Compute for each module $m$ the number of upgrade dependencies, upgrading $m$ from version $i$ to latest version.

3. Visualize results:
   - Heat maps: high level overview
   - Dependency graphs: low level details of a particular upgrade
Find minimal number of upgrade dependencies

Goal: upgrade A from version 1 to 3

<table>
<thead>
<tr>
<th>Module</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>X</td>
</tr>
</tbody>
</table>

Upgrade A only
Compatible?
No missing deps?

<table>
<thead>
<tr>
<th>Module</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>X</td>
</tr>
</tbody>
</table>

Upgrade B

<table>
<thead>
<tr>
<th>Module</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>X</td>
</tr>
</tbody>
</table>

Upgrade C

<table>
<thead>
<tr>
<th>Module</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
</tbody>
</table>

Upgrade C
High level visualization: heat map

- Each cell indicates complexity for each module (rows) from an older version (column) to the latest version.

- Darker means more upgrade dependencies (hence, more complex).
Low level visualization: upgrade dependency graphs

• Dependency graphs provide more details about a particular upgrade

• **Vertices**: all modules involved with upgrade.

• **Edge** from module $n_1$ to $n_2$ iff there is an upgrade dependency from $n_1$ to $n_2$.
  • **Green edge**: upgrade dependency due to symbol addition(s).
  • **Red edge**: upgrade dependency due to symbol removal(s).
  • **Black edge**: green and red combined.

• **Thickness**: number of symbols.
Case study performed at ASML Netherlands B.V.

- Manufacturer of chip-making equipment.
- Designs, develops and integrates systems to produce semiconductors.

- 1000 software developers
- 40 MLOC
- 327 modules
- 7000 interfaces
- 9 versions of the software analyzed (between Oct. ’11 – Jul. ’12)
Result (1)

- Applied approach on software at ASML.
- 327 modules * 8 versions = **2616 upgrade scenarios**
- Processing time: **16 hours** for all scenarios
  - With limitations on search space.
• Upgrade dependencies due to removal of symbols

• What if we ignore the removal of symbols?
Conclusion

Synchronizing patches is time-consuming and error-prone.

Module-oriented patching: modules should be independent.

Determine upgrade dependencies for each module.

Case study: provided new insights in how modules are related and how to improve future upgrades.