Patl: Safe Program Transformation between APIs with Many-to-Many Mappings

Yingfei Xiong

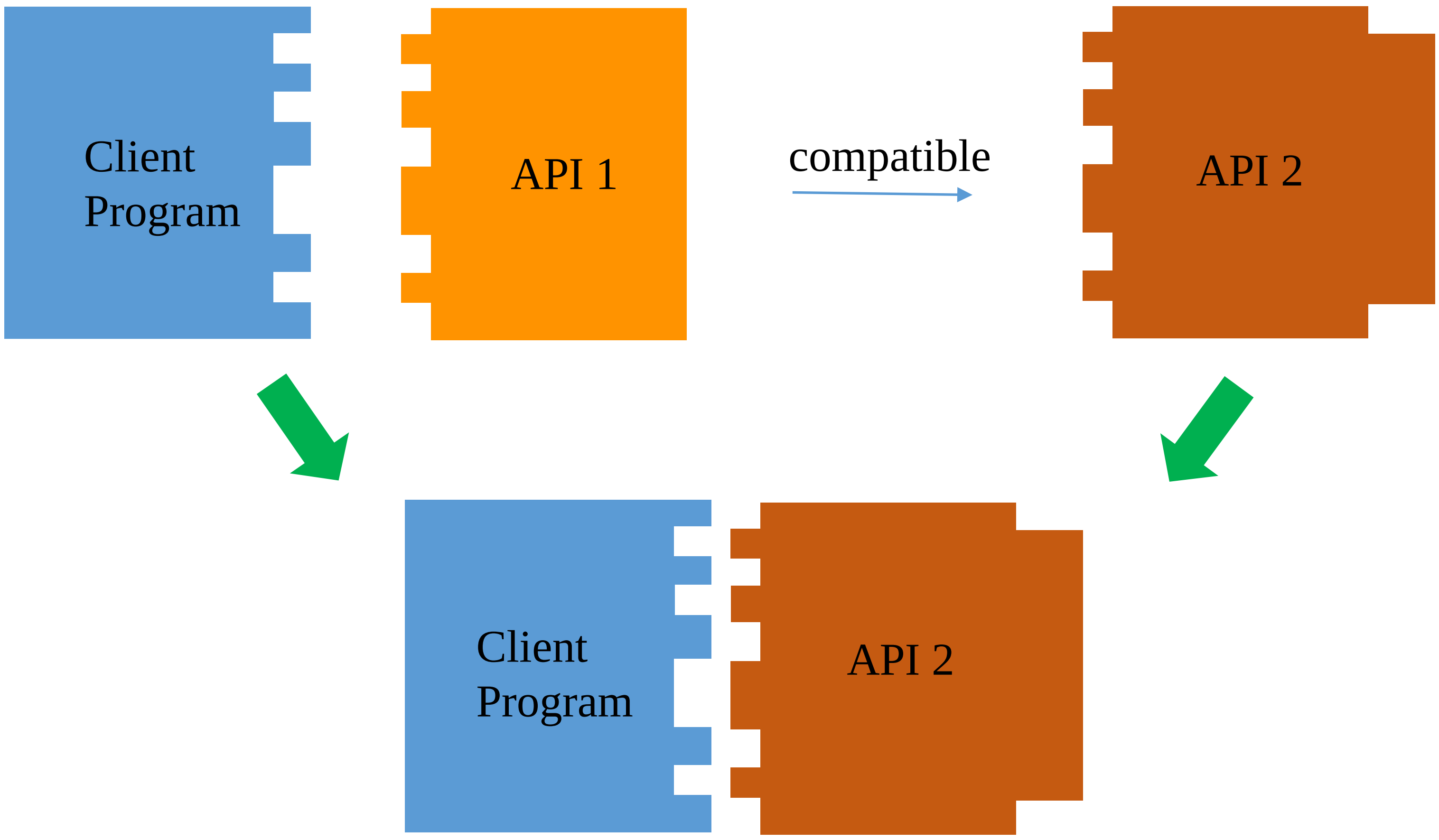
Peking University

API change is common!

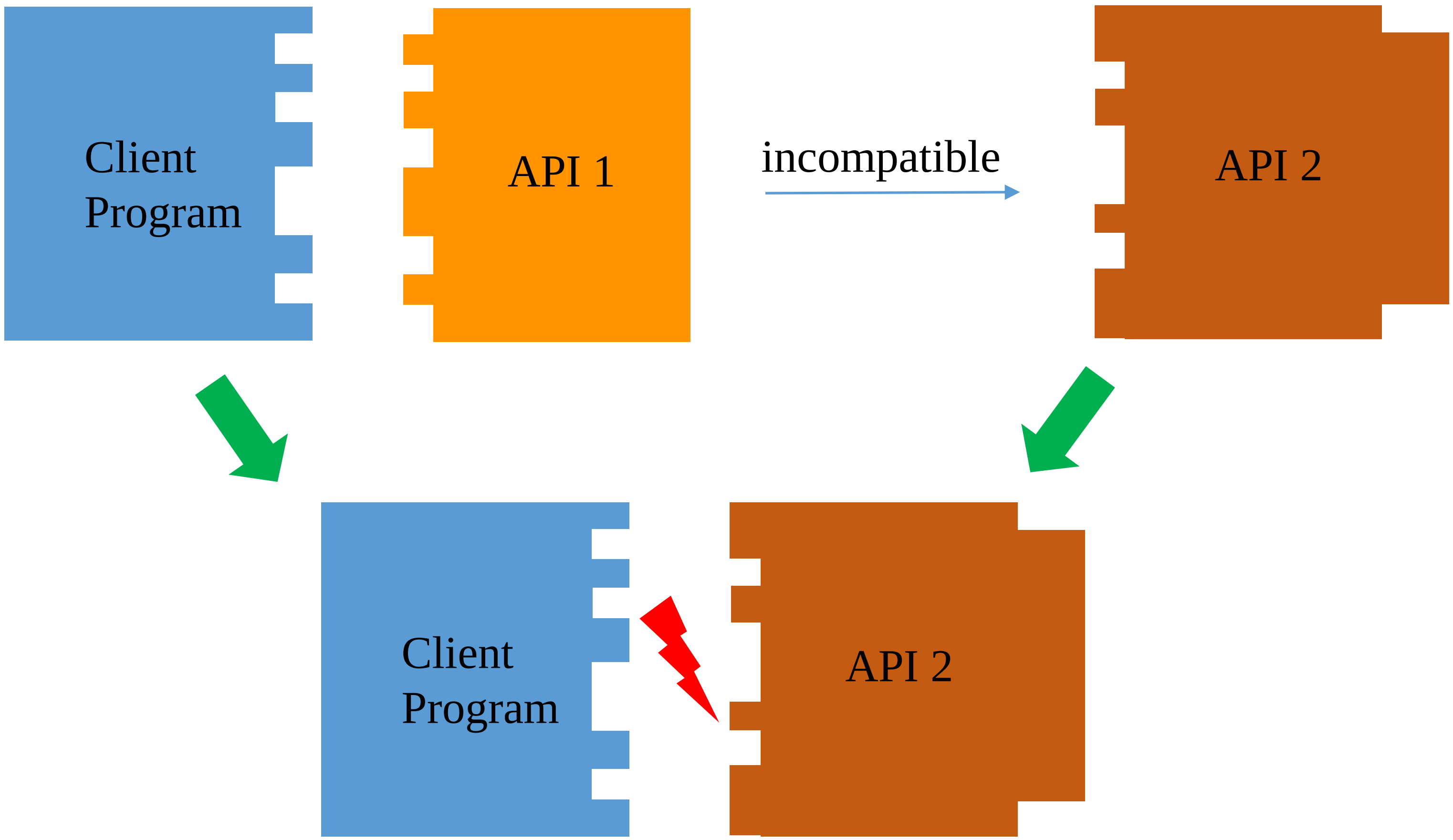
- New APIs are released to replace older ones.
- Migrate a program to another platform.
- Discontinued API support.

Switch the use of old APIs to new ones!

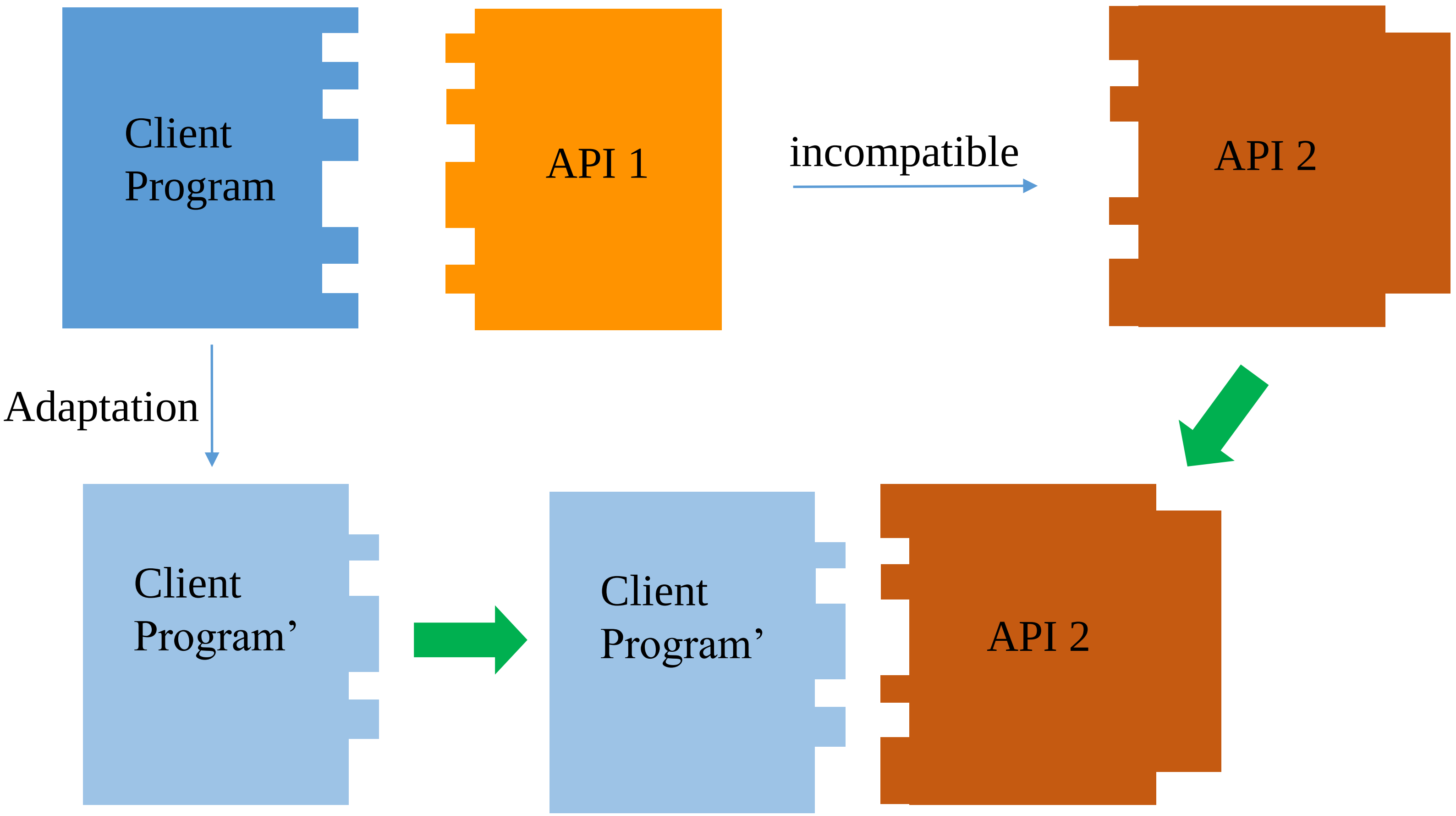
Compatible API change



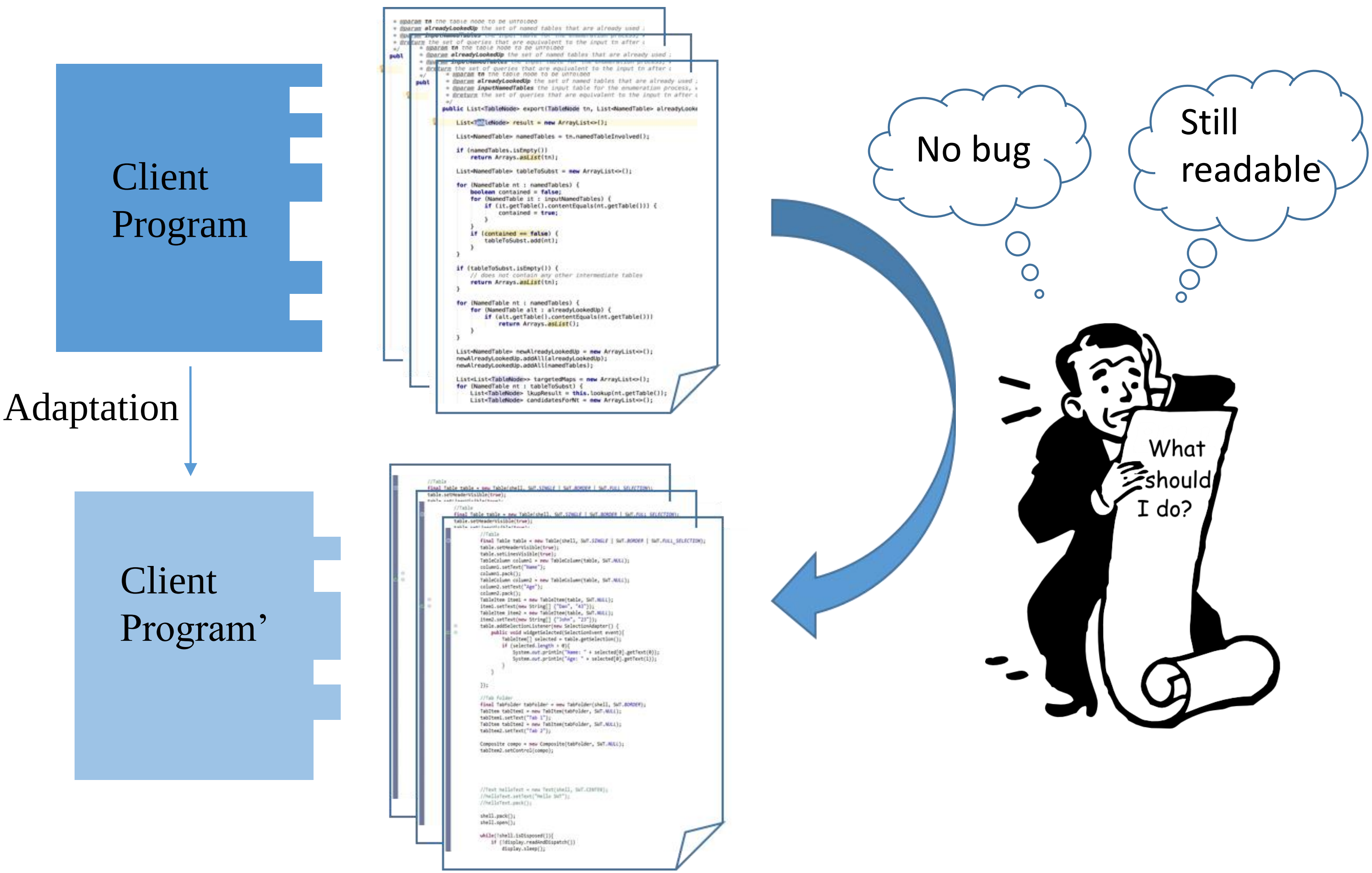
Incompatible API change



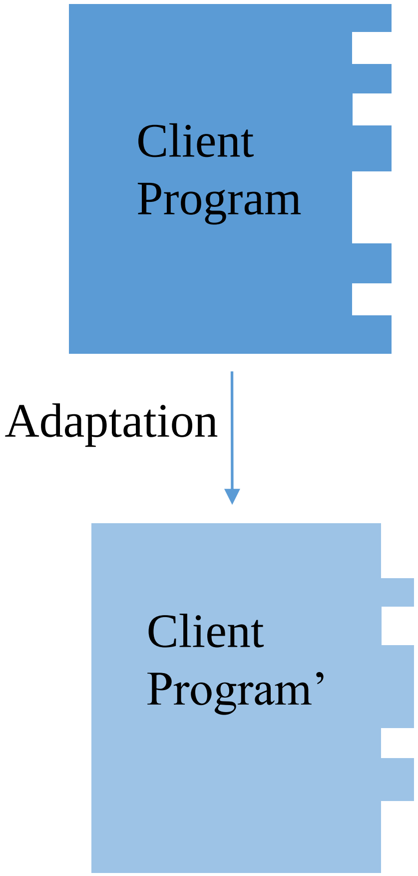
Solution: Adapting Client Program



Adapting Client Program: Not easy



Adapting Client Program



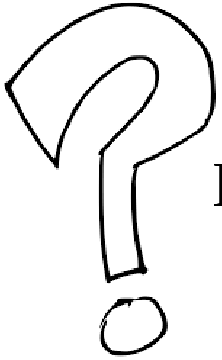
- API users view:
  - A tool to automatically adapt the client program.
  - Adapted programs should be correct and readable.
- API developers view:
  - How to develop such transformation tools easily?

Transformation languages support:

- Specify transformation with rules.

Existing language support

- General purpose transformation languages:
  - Stratego [1], TXL [2]
  - Pro: expressive
  - Con: not specialized for API adaptation task, low-level
- API adaptation domain specific languages:
  - SWIN [3], Twinning [4]
  - Pro: specialized, ease-to-use
  - Con: captures only one-to-many mappings, less expressive



How can we help support many-to-many transformations?

Many-to-Many (M-to-M) transformation

- Definition: Match a sequence of statements in the source program, substitute them with another sequence of statements.
- E.g. (Swing to SWT)

```
//rule rButton
(jb: JButton->Button,
 parent: JPanel->Composite) {
 - jb = new JButton();
 - parent.add(jb);
 + jb = new Button(parent, SWT.PUSH);
}
```

- Basic transformation: match and substitute.

```
jb = new JButton();
parent.add(jb);
```

→

```
jb = new Button(parent, SWT.PUSH);
```

M-to-M transformation

```
//rule rButton
(jb: JButton->Button,
 parent: JPanel->Composite) {
 - jb = new JButton();
 - parent.add(jb);
 + jb = new Button(parent, SWT.PUSH);
}
```

```
//Case 1:
jb = new JButton();
if(parent != null) {
  parent.add(jb);
}
```

```
//Case 2:
jb = new JButton();
s = jb.getUIClassID(();
parent = new JPanel();
parent.add(jb);
```

```
//Case 3:
jb = new JButton();
defaultButton = jb;
parent.set(defaultButton);
```

Challenge:

The source sequence can appear in many different forms in the client program.

- Match them with the rules. ☺ (SmPL [5])
- Transform them safely. ☹ → ☺ (Our approach)

Insight: guided-normalization

```
//rule rButton
(jb: JButton->Button,
 parent: JPanel->Composite) {
 - jb = new JButton();
 - parent.add(jb);
 + jb = new Button(parent, SWT.PUSH);
}
```

```
//Case 1:
jb = new JButton();
if(parent != null) {
  parent.add(jb);
}
```

```
//Case 2:
jb = new JButton();
s = jb.getUIClassID(();
parent = new JPanel();
parent.add(jb);
```

```
//Case 3:
jb = new JButton();
defaultButton = jb;
parent.set(defaultButton);
```

↓

```
//Case 1':
if(parent != null) {
  jb = new JButton();
  parent.add(jb);
}
```

```
//Case 2':
parent = new JPanel();
jb = new JButton();
parent.add(jb);
s=jb.getUIClassID(();
```

```
//Case 3':
jb = new JButton();
parent.set(jb);
defaultButton = jb;
```

Transformation rule writer only need to consider basic transformation!!

Guided-normalization

- Normalize the source program
  - Semantics-preserving.
  - Touch less unrelated statements.
  - Matched statements appear consecutively after normalization.
- Preliminary: Program analysis
  - Analyze dependency and alias relations in the program to ensure normalization will not go wrong.

Our transformation pipeline: Patl

$$\Pi, p \xrightarrow{\text{Match}} M$$

$$p \xrightarrow[\text{Guided normalization}]{} p' \xrightarrow[\text{Basic transformation}]{M} p''$$

Π: transformation rules.
p: client program to be transformed.
M: match instances
p': normalized program.
p'': transformed program with new API use.

```
//Case 1:
jb = new JButton();
if(parent != null) {
  parent.add(jb);
}
```

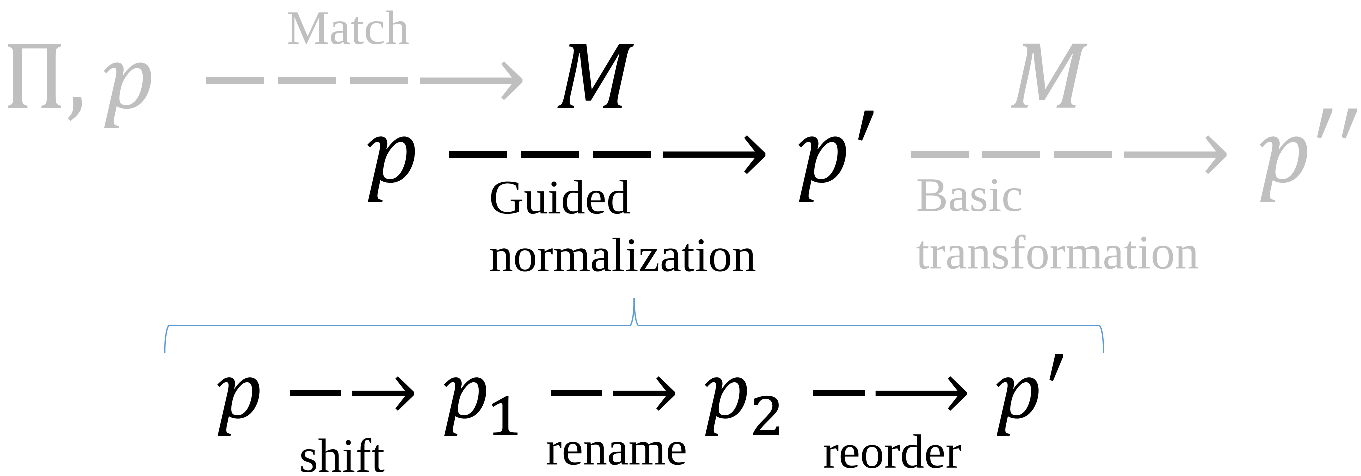
⟶

```
//Case 1':
if(parent != null) {
  jb = new JButton();
  parent.add(jb);
}
```

⟶

```
//Case 1':
if(parent != null) {
 jb = new Button(parent, SWT.PUSH);
}
```

Guided-Normalization



- Guided-shift
  - Make statements matched by a rule appear in the same block.
- Guided-rename
  - Make aliases in these statements have the same name.
- Guided-reorder
  - Make matched statements appear consecutively

Guided-normalization: example

```
//rule rButton
(jb: JButton->Button,
 parent: JPanel->Composite) {
 - jb = new JButton();
 - parent.add(jb);
 + jb = new Button(parent, SWT.PUSH);
}
```

```
btn = new JButton();
btn.setAlignmentX(alX);
System.out.print(alX);
b = panel != null;
if (b) {
  panel.add(btn);
} else {
  defaultBtn = btn;
  defaultPnl.add(defaultBtn);
}
```

normalize →

```
System.out.print(alX);
b = panel != null;
if (b) {
  btn = new JButton();
  panel.add(btn);
  btn.setAlignmentX(alX);
} else {
  x = new JButton();
  defaultPnl.add(x);
  btn = x;
  defaultBtn = btn;
  btn.setAlignmentX(alX);
}
```

Guided-shift

- If matched statements appear in different blocks, shift them into basic blocks.

```java
btn = new JButton();
btn.setAlignmentX(alX);
System.out.print(alX);
b = panel != null;
if (b) {
  panel.add(btn);
} else {
  defaultBtn = btn;
  defaultPnl.add(defaultBtn);
}
```

GuidedShift →

```java
System.out.print(alX);
b = panel != null;
if (b) {
  btn = new JButton();
  btn.setAlignmentX(alX);
  panel.add(btn);
} else {
  btn = new JButton();
  btn.setAlignmentX(alX);
  defaultBtn = btn;
  defaultPnl.add(defaultBtn);
}
```

Guided-Rename

- Aliases in matched statements are renamed to have the same names.

```
System.out.print(alX);
b = panel != null;
if (b) {
  btn = new JButton();
  btn.setAlignmentX(alX);
  panel.add(btn);
} else {
  btn = new JButton();
  btn.setAlignmentX(alX);
  defaultBtn = btn;
  defaultPnl.add(defaultBtn);
}
```

GuidedRename →

```
System.out.print(alX);
b = panel != null;
if (b) {
  btn = new JButton();
  btn.setAlignmentX(alX);
  panel.add(btn);
} else {
  x = new JButton();
  btn = x;
  btn.setAlignmentX(alX);
  defaultBtn = btn;
  defaultPnl.add(x);
}
```

Guided-reorder

- Reorder matched statements so that they appear consecutively.

```
System.out.print(alX);
b = panel != null;
if (b) {
  btn = new JButton();
  btn.setAlignmentX(alX);
  panel.add(btn);
} else {
  x = new JButton();
  btn = x;
  btn.setAlignmentX(alX);
  defaultBtn = btn;
  defaultPnl.add(x);
}
```

Rename

```
System.out.print(alX);
b = panel != null;
if (b) {
  btn = new JButton();
  panel.add(btn);
  btn.setAlignmentX(alX);
} else {
  x = new JButton();
  defaultPnl.add(x);
  btn = x;
  defaultBtn = btn;
  btn.setAlignmentX(alX);
}
```

Guided-normalization: Safety

- How to ensure normalization is semantics-preserving?
- Semantics-preserving transformation primitives:
  - Primitive shift
  - Primitive swap
  - Primitive left-value renaming
  - Primitive right-value renaming
  - Fresh-variable introducing

Commonly used in compiler optimization

- Safety: guided-normalization can be decomposed into transformation primitives. (Proof in the paper!)

Warnings in transformation

- Guided-normalization is not always applicable:
  - Dependency may be violated.

```
x = y.a();
if (x) {
    y.b(x);
}
```

```
if (x) {
    x = y.a();
    y.b(x);
}
```

```
x = y.a();
Send(x, y);
y.b(x);
```

```
x = y.a();
y.b(x);
Send(x, y);
```

- Our system will generate warnings in such cases rather than silently making mistakes.

Evaluation

- Q1: How important is guided-normalization in transforming programs between APIs?
- Q2: How many cases cannot be handled by our approach?
- Q3: How many warnings will be generated in real world cases?

Evaluation: set-up

- Three real-world cases:
  - Jdom → Dom4J
  - Google calendar v2 → v3
  - Swing → SWT
- Six open source projects using these APIs.

Client	KLOC	Classes	Methods	Case
husacct	195.6	1187	5977	Jdom/Dom4j
serenoa	12.2	52	523	Jdom/Dom4j
openfuxml	112.5	727	4098	Jdom/Dom4j
clinicaweb	3.9	74	213	Calendar
blasd	9.7	199	729	Calendar
goofs	8.6	78	643	Calendar
evochamber	12.8	132	868	Swing/SWT
swingheat	2.3	30	186	Swing/SWT
marble	1.6	10	56	Swing/SWT
Total	359.2	2489	13293	–

Evaluation: result

Client	CF	CL	W	U	I	MM	GN
husacct	42	852(100%)	0(0%)	0(0%)	0	0(0%)	0(0%)
serenoa	8	273(98.9%)	0(0%)	3(1.1%)	0	9(3.3%)	0(0%)
openfuxml	72	983(94.8%)	0(0%)	54(5.2%)	15	2(0.2%)	0(0%)
clinicaweb	5	81(100%)	0(0%)	0(0%)	8	34(42%)	0(0%)
blasd	5	26(63.4%)	8(19.5%)	7(17.1%)	0	13(50%)	2(15.4%)
goofs	13	100(80.0%)	12(9.6%)	13(10.4%)	27	27(27%)	0(0%)
evochamber	9	587(98.3%)	10(1.7%)	0(0%)	0	330(56.2%)	109(33.0%)
swingheat	21	653(100%)	0(0%)	0(0%)	0	461(70.6%)	394(85.5%)
marble	6	488(98.6%)	0(0%)	7(1.4%)	0	240(49.2%)	220(91.7%)
Total	181	4043(97.3%)	30(0.7%)	84(2.0%)	50	1116(27.6%)	725(65.0%)

CF = number of changed files, CL= number of changed lines, percentages in CL = CL / (CL+W+U), W = the number of lines of code that have warnings, percentages in W = W / (CL+W+U), U = number of lines that PATL cannot transform, percentages in U = U / (CL+W+U), I = number of lines impossible to transform, MM = number of lines that are involved in many-to-many mappings, percentages in MM = MM / CL, GN = number of lines that require guided normalization, percentages in GN = GN / MM.

Evaluation: result

Transformation Rules

Transformation	Rules	Classes	Methods	M-to-m
Jdom/Dom4j	84	12	77	12(14.3%)
Calendar	42	14	45	21(50.0%)
Swing/SWT	110	40	82	54(49.1%)
Total	236	66	204	87(36.9%)

Empirically, Patl is ease-to-use! ☺

An example not handled by Patl

```
XMLOutputter out = new XMLOutputter();
for (Element e : rulesElements)
  rulesToRegister += out.outputString(e);
```

Involves transforming across a ‘for’ statement

```
StringWriter sw = new StringWriter();
XMLWriter out = new XMLWriter(sw);
for (Element e : rulesElements)
  out.write(e);
rulesToRegister += sw.toString();
```

Limitations

- Solves only statement level transformation, not class level transformation.
  - E.g. Inheritance from an old API class.
- Does not model synchronization in transformation.
  - E.g. A method may change from synchronized to unsynchronized.

Conclusion

- Guided-normalization helps enhance transformation language support to solve M-to-M transformation programs in API adaptation.
- Guided-normalization:
  - Safe: semantics-preserving.
  - Help ease transformation tool developing.

References

[1] Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. K. Olmos and E. Visser. In *CC*, pages 204–220, 2005.

[2] The TXL Source Transformation Language. J. R. Cordy. *Sci. Comput. Program.*, pages 190–210, 2006.

[3] Using Twinning to Adapt Programs to Alternative APIs. M. Nita and D. Notkin. In *ICSE*, pages 205–214, 2010.

[4] SWIN: Towards Type-Safe Java Program Adaptation Between APIs. J. Li, C. Wang, Y. Xiong, and Z. Hu. In *PEPM*, pages 91–102, 2015.

[5] Documenting and Automating Collateral Evolutions in Linux Device Drivers. Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. In *EuroSys*, pages 247–260, 2008.