TRANSPILATION USING RECURSIVE REWRITE RULES FROM LEGACY TO MAINTAINABLE CODE

Tristan Albers, Jos Hegge, **Piërre van de Laar**, | 040coders
Niels Brouwers, Paul Nelissen,
Wilbert Alberts, George Azis, Theo Baan, Danny Handoko, Quint van der Linden

ASML

Capgemini engineering

TNO ESI
Powered by industry and academia

ABOUT US

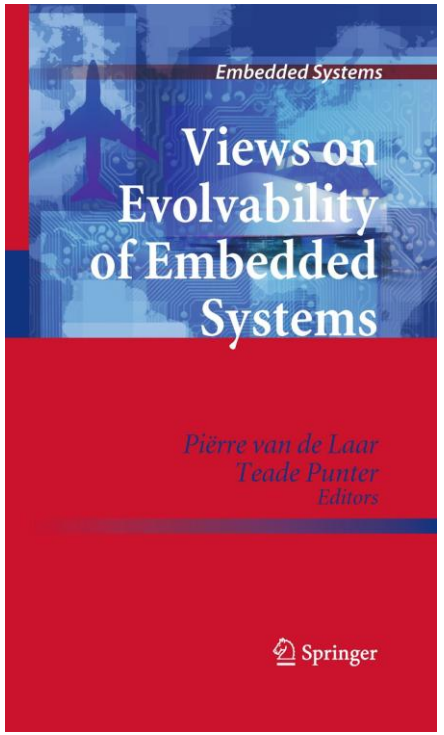
- TNO-ESI is an open and diverse innovation research centre with strong partnerships with industry-leading high-tech companies

- Capgemini is a leading advisor and transformation partner to companies around the world

- ASML is one of the world's leading manufacturers of chip-making equipment

ABOUT ME – PIËRRE VAN DE LAAR

- Industrial innovator researching evolving product families



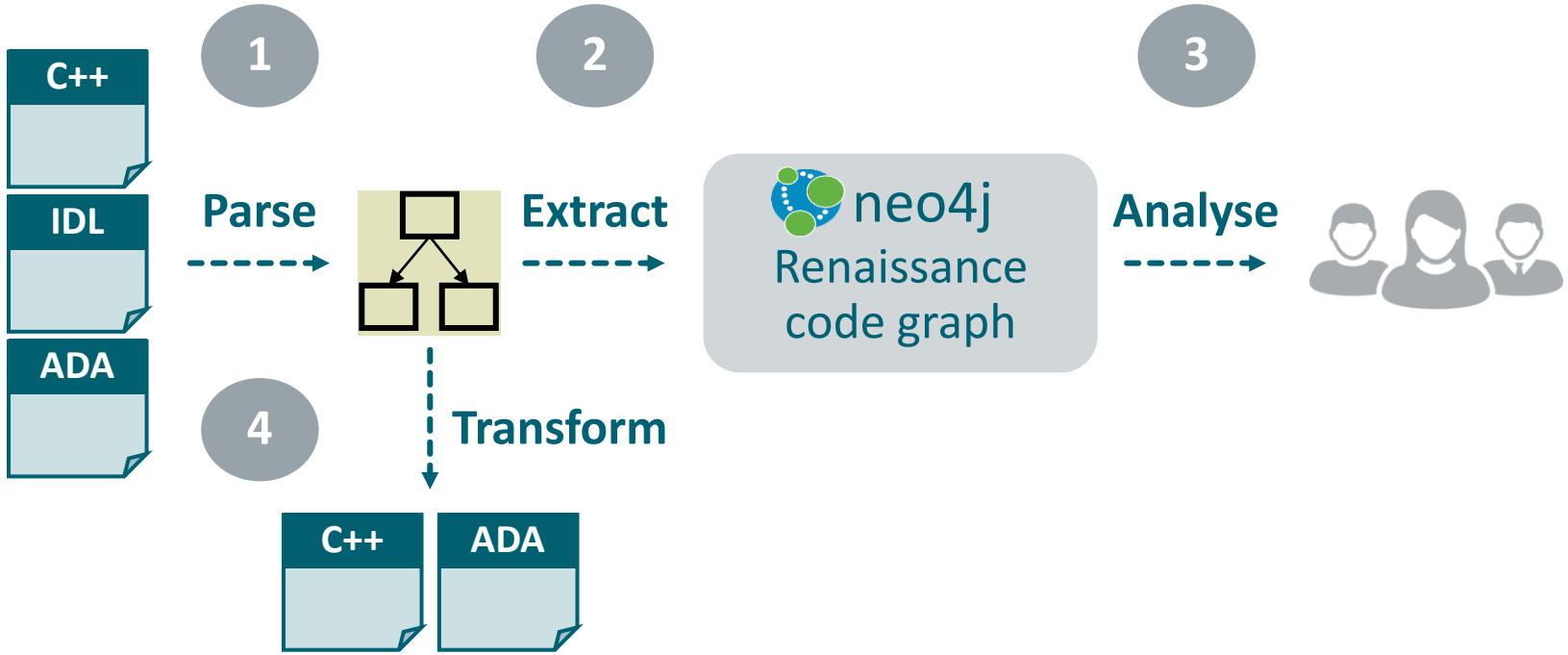
- Passionate about architecture, design, and code quality

- Wants to help the young software community to move from green field to brown field development

1

RENAISSANCE

RENAISSANCE METHODOLOGY



RENAISSANCE METHODOLOGY

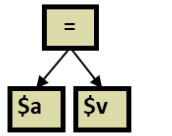
1

Parse

PATTERN

```
int $a = $v;
```

parse



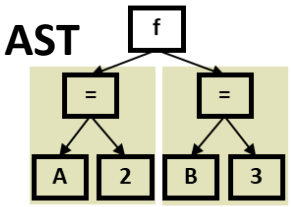
match

CODE

```
void f(){
    int a = 2;
    int b = 3;
}
```

parse

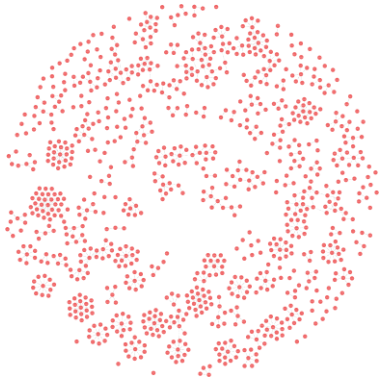
AST



2

Extract

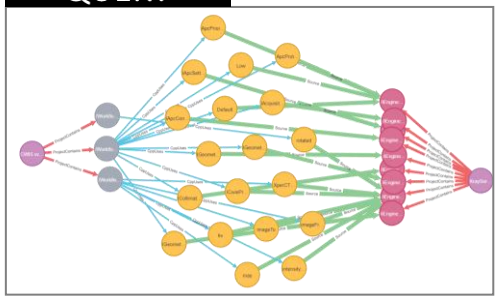
neo4j



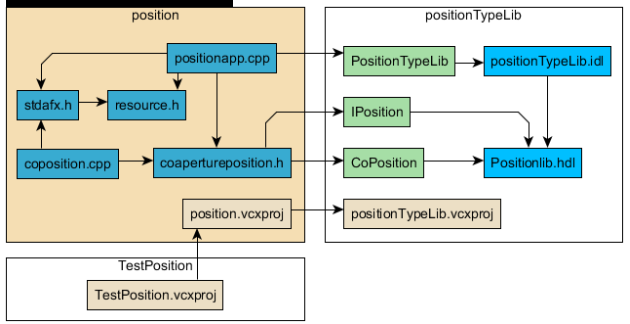
3

Analyse

QUERY



ARCH VIEWS



4

Transform

FLUENT

```
transformation {
    action.find(BEFORE)
    .replace(AFTER)
    ...
}
```

BEFORE

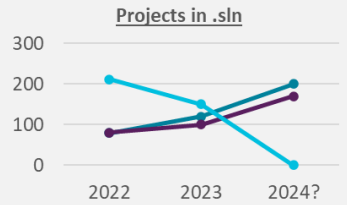
```
void FooTest::TestMethod(){
    CPPASSERT(avariable);
}
```

AFTER

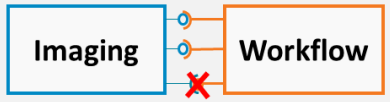
```
TEST_F(FooTest,TestMethod){
    EXPECT_TRUE(avariable);
}
```

DASHBOARD

Track progress



Measure hotspots



Track arch Violations

EVOLVING PROGRAMMING LANGUAGES (1/2)

- Ada 2012 introduced quantified expressions

```
type Integer_Arr is
  array (Positive range <>) of Integer;

function Is_Zero (A : Integer_Arr)
                  return Boolean is
   (for all I in A'Range => A (I) = 0);

function Has_Zero (A : Integer_Arr)
                   return Boolean is
   (for some I in A'Range => A (I) = 0);
```

- In 2020, developers still wrote loops instead of using quantified expressions

```
function Is_Zero (A : Integer_Arr)
                  return Boolean is
begin
   for I in A'Range loop
      if A (I) /= 0 then
         return False;
      end if;
   end loop;

   return True;
end Is_Zero;
```

```
function Has_Zero (A : Integer_Arr)
                   return Boolean is
begin
   for I in A'Range loop
     if A (I) = 0 then
        return True;
     end if;
   end loop;

   return False;
end Has_Zero;
```

EVOLVING PROGRAMMING LANGUAGES (2/2)

- Not just a warning!
- Find & replace pattern
  - Simplify
    - not ($var /= $expr) → $var = $expr
    - … is begin return $expr; end …; → … is $expr
- Educated developers
  - Usage of quantified expressions increased
  - More readable code due to higher abstraction
  - More idiomatic Ada

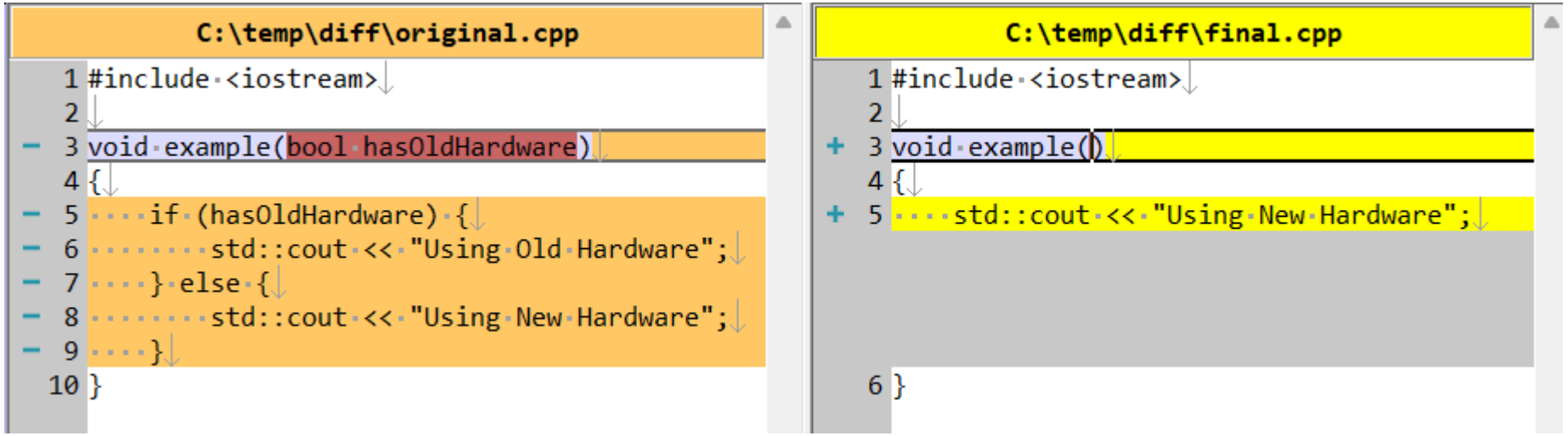
```
Rewriter_For_All_Range_Return :
  aliased constant Rewriter_Find_And_Replace :=
  Make_Rewriter_Find_And_Replace
    (Make_Pattern
       ("for $S_I in $S_Range " &
        "loop if $S_Cond then return false; end if; end loop; " &
        "return true;",
        Stmts_Rule),
     Make_Pattern
       ("return (for all $S_I in $S_Range => not ($S_Cond));",
        Return_Stmt_Rule));
```

```
function Is_Zero (A : Integer_Arr)
                  return Boolean is
begin
   for I in A'Range loop
      if A (I) /= 0 then
         return False;
      end if;
   end loop;

   return True;
end Is_Zero;
```

```
function Is_Zero (A : Integer_Arr)
                  return Boolean is
  (for all I in A'Range => A (I) = 0);
```

HOW TO REALIZE A TRANSFORMATION?

- Complete transformation



C:\temp\diff\original.cpp

```
 1 #include <iostream>
 2
 3 void example(bool hasOldHardware)
 4 {
 5     if (hasOldHardware) {
 6         std::cout << "Using Old Hardware";
 7     } else {
 8         std::cout << "Using New Hardware";
 9     }
10 }
```

C:\temp\diff\final.cpp

```
 1 #include <iostream>
 2
 3 void example()
 4 {
 5     std::cout << "Using New Hardware";
 6 }
```

- Hard for reviewer to see
  - What was the overall approach?
  - What was really required? What is unique for this transformation?
  - What are consequences, such as simplifications and changes in indentation? What is reused?

HOW TO REALIZE A TRANSFORMATION? IN STEPS!

Recipe

- Old hardware no longer available
- Simplification
- Correct indentation

C:\temp\diff\original.cpp

```
1  #include <iostream>
2
3  void example(bool hasOldHardware)
4  {
5      if (hasOldHardware) {
6          std::cout << "Using Old Hardware";
7      } else {
8          std::cout << "Using New Hardware";
9      }
10 }
```

C:\temp\diff\minimal.cpp

```
1  #include <iostream>
2
3  void example()
4  {
5      if (false) {
6          std::cout << "Using Old Hardware";
7      } else {
8          std::cout << "Using New Hardware";
9      }
10 }
```

C:\temp\diff\minimal.cpp

```
1  #include <iostream>
2
3  void example()
4  {
5      if (false) {
6          std::cout << "Using Old Hardware";
7      } else {
8          std::cout << "Using New Hardware";
9      }
10 }
```

C:\temp\diff\intermediate.cpp

```
1  #include <iostream>
2
3  void example()
4  {
5          std::cout << "Using New Hardware";
6  }
```

C:\temp\diff\intermediate.cpp

```
1  #include <iostream>
2
3  void example()
4  {
5          std::cout << "Using New Hardware";
6  }
```

C:\temp\diff\final.cpp

```
1  #include <iostream>
2
3  void example()
4  {
5      std::cout << "Using New Hardware";
6  }
```

LESSONS LEARNED

- Use recipes
  - Use basic rules – find, filter, and replace – separation of concern
  - Organize rules – repeat and sequence
- Support reviewer
  - Recipe, complete transformation, and the individual steps
  - Focus on the risky parts
- Enable reuse
  - Improves quality
  - Simplifications are typically needed
  - Filtering based on the variables read and written within code snippets

2

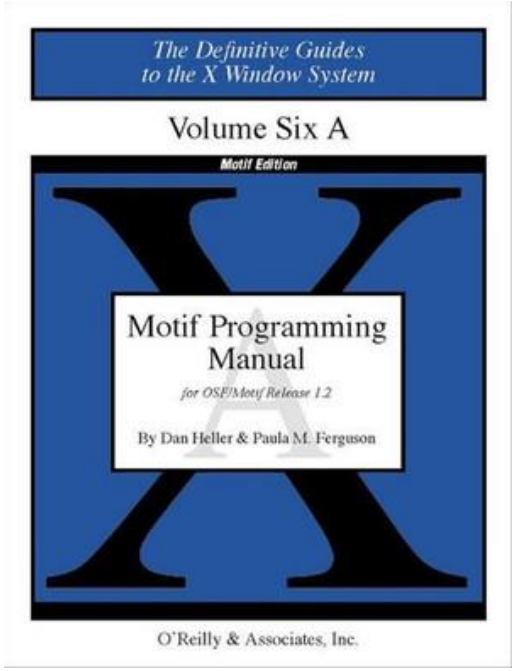
USE CASE @ ASML

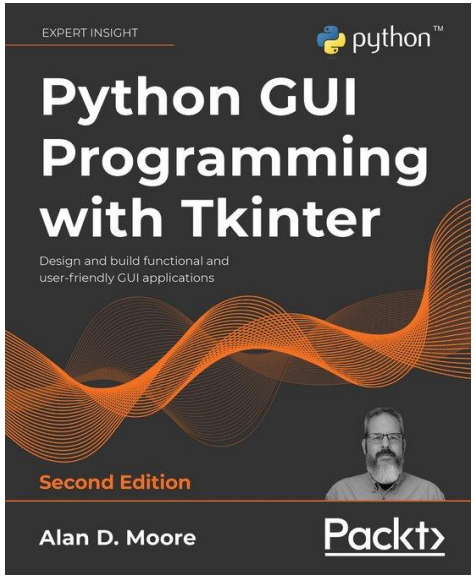
SOFTWARE STACK @ ASML

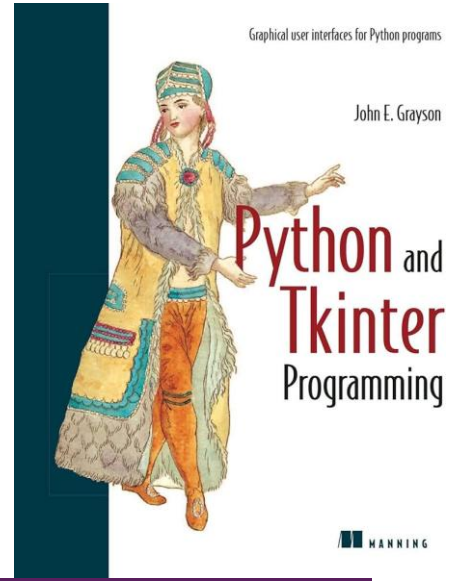
- Two programming languages C and Python
- Multiple components

- Code
  - Well-structured
  - Readable and maintainable

- Two UI frameworks
  - Motif in C
  - Tkinter in Python

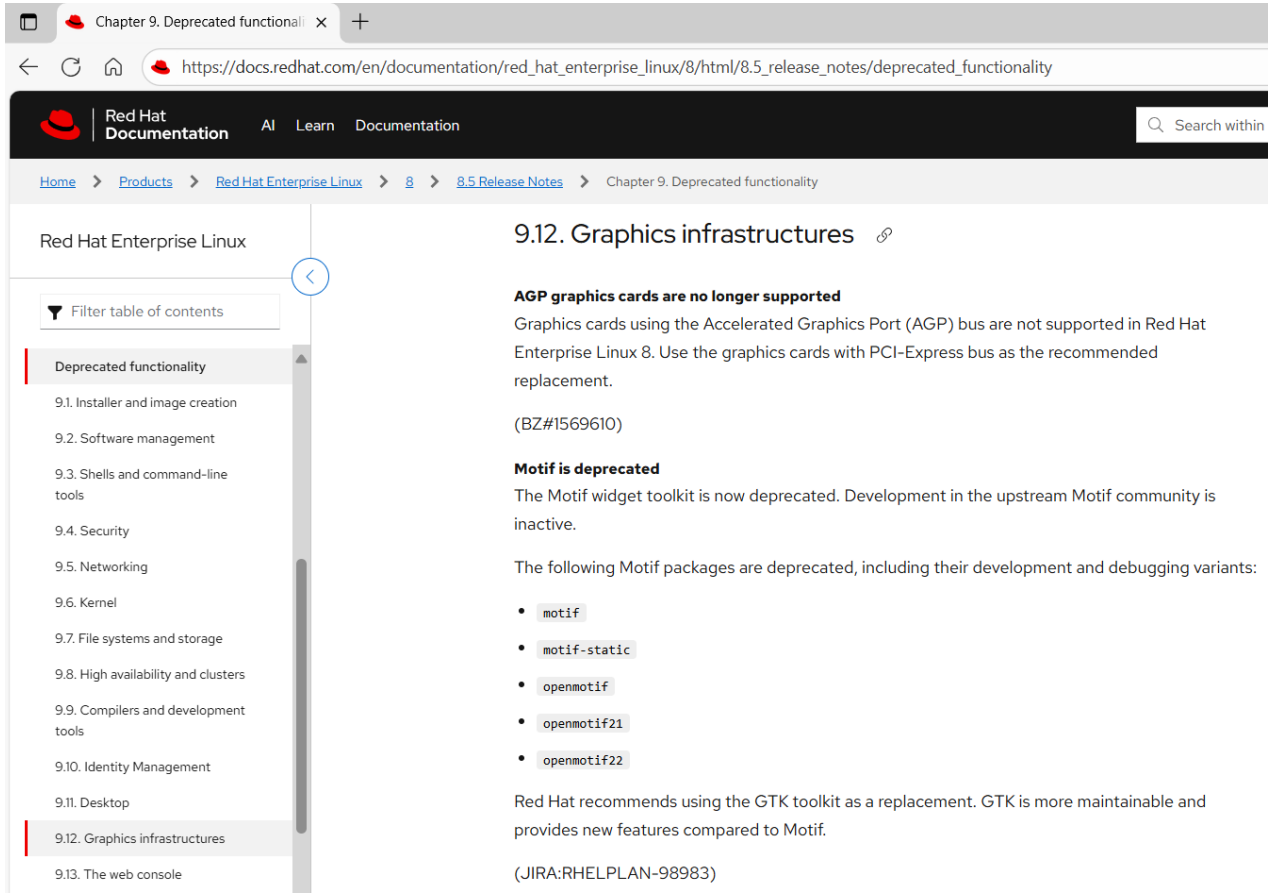






TRIGGER FOR USE CASE

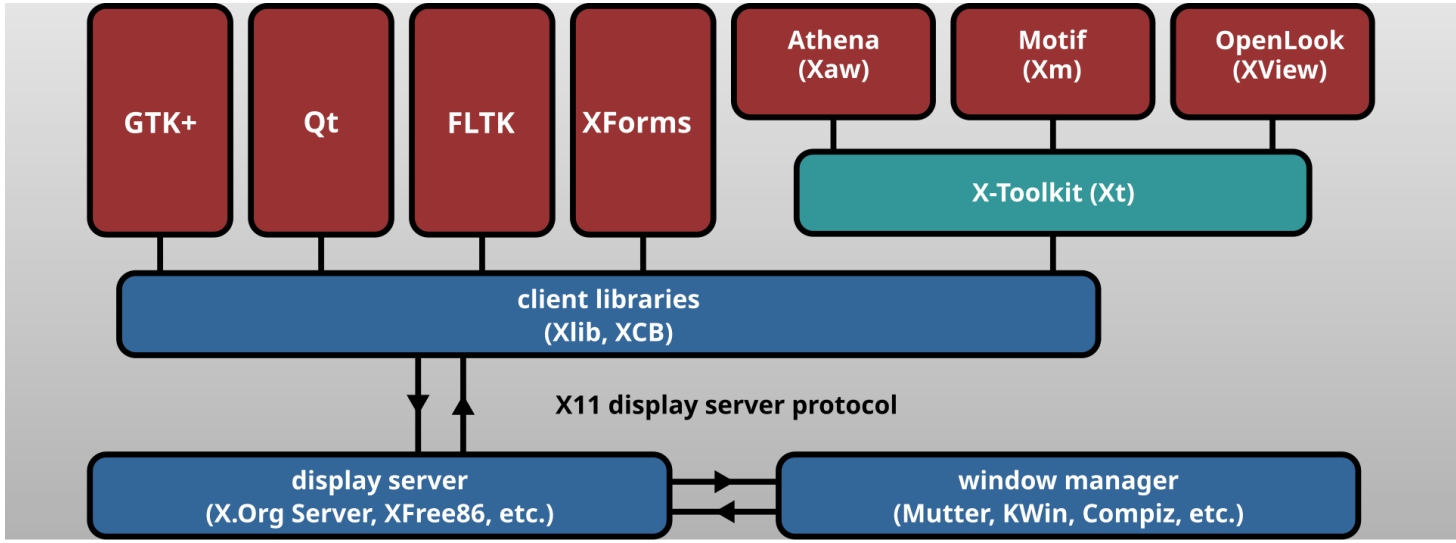
Motif is deprecated



CHOICES

- ASML maintains Motif themselves

- Migrate to another UI framework in C
  - GTK, recommended by Red Hat
  - Qt
  - …



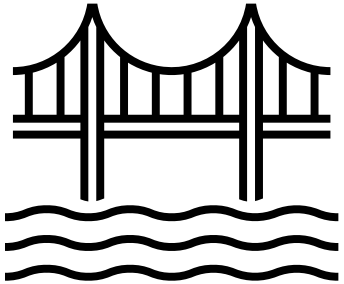
- Migrate to Tkinter UI framework in Python

transpilation

TRANSFORMATION VS TRANSPILATION

- When transforming code, source and target language are the same
- Incremental way-of-working
- Many steps – separation of concerns

- When transpiling code, source and target language differ
- Impossible to parse intermediate result
  i.e., partly in source and partly in target language
- Single step – integrated effort



explainability

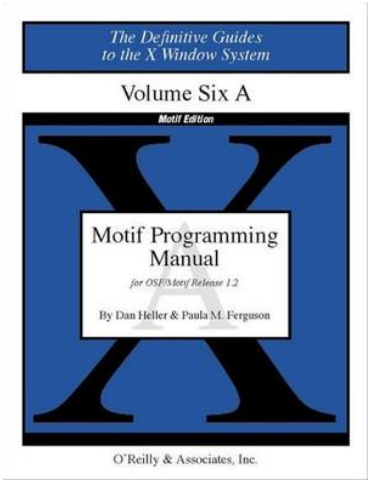
DESIRED SOLUTION

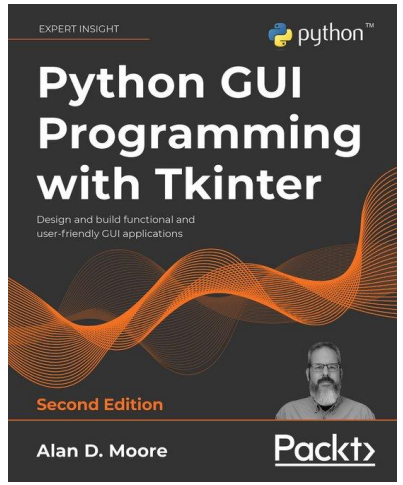
from

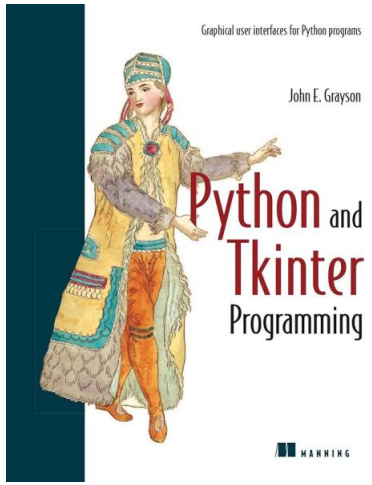
Motif in C

to

Tkinter in Python







The resulting code should be acceptable for the developers

readable & maintainable
explainable transformation

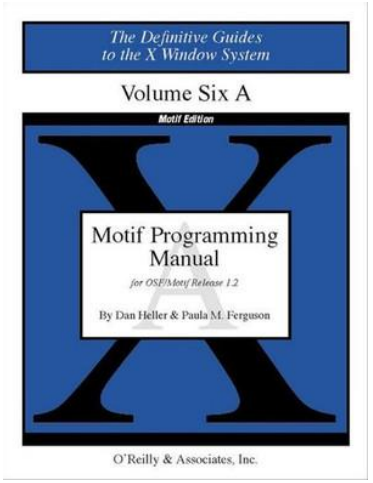
DESIRED SOLUTION

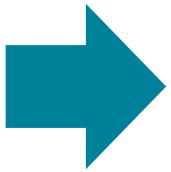
from

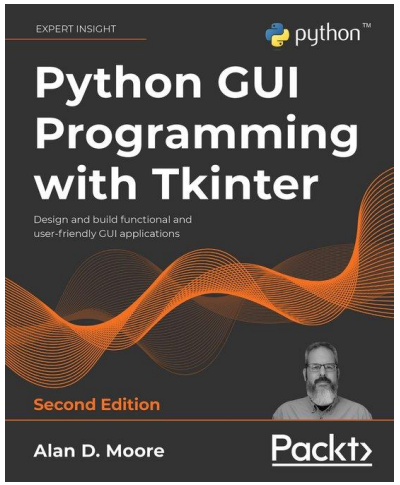
Motif in C

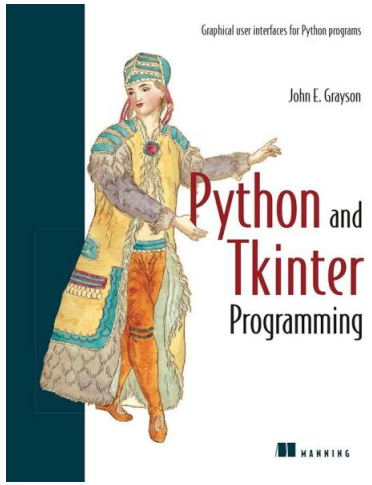
to

Tkinter in Python







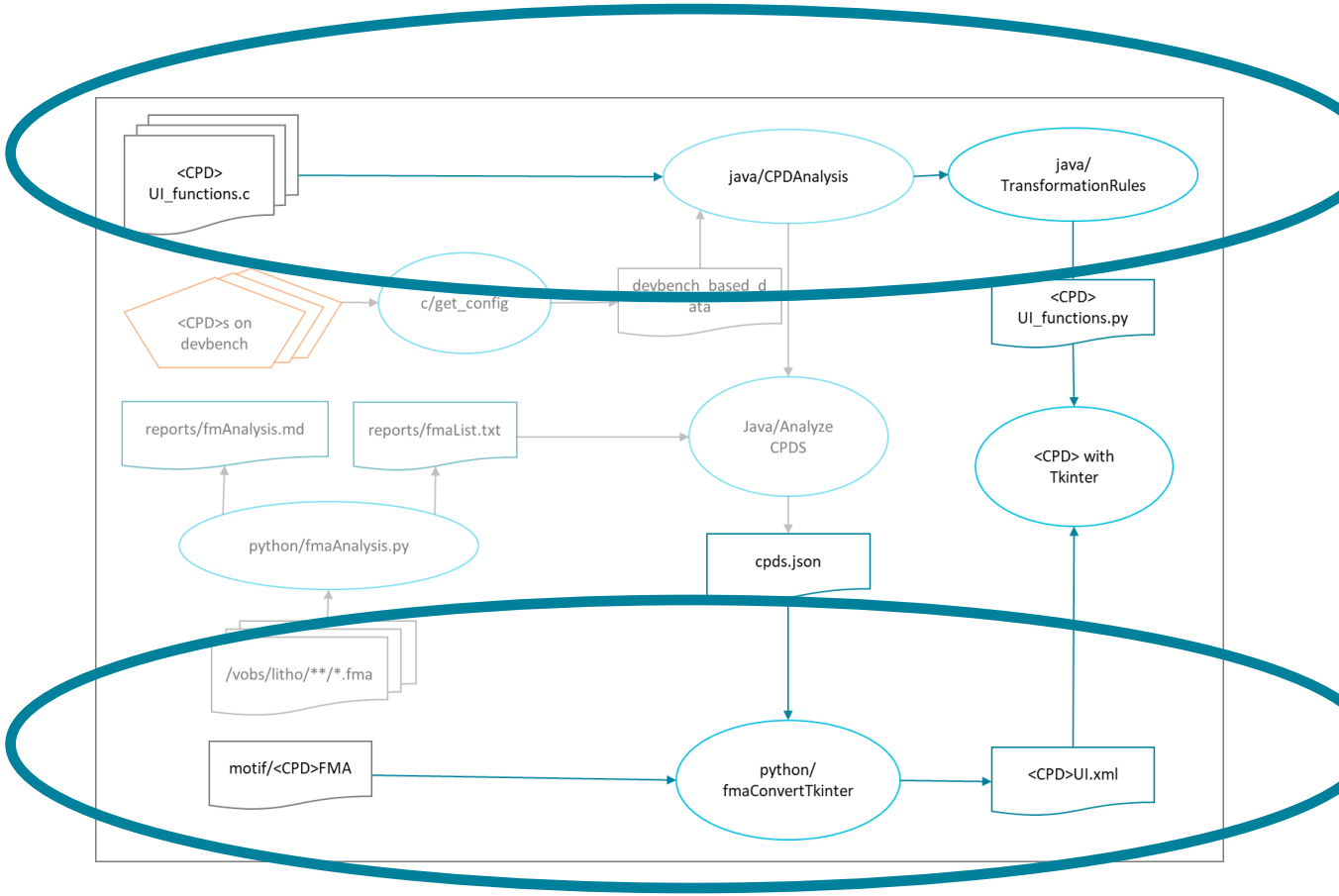


Estimation for manual transpilation

100 man-years

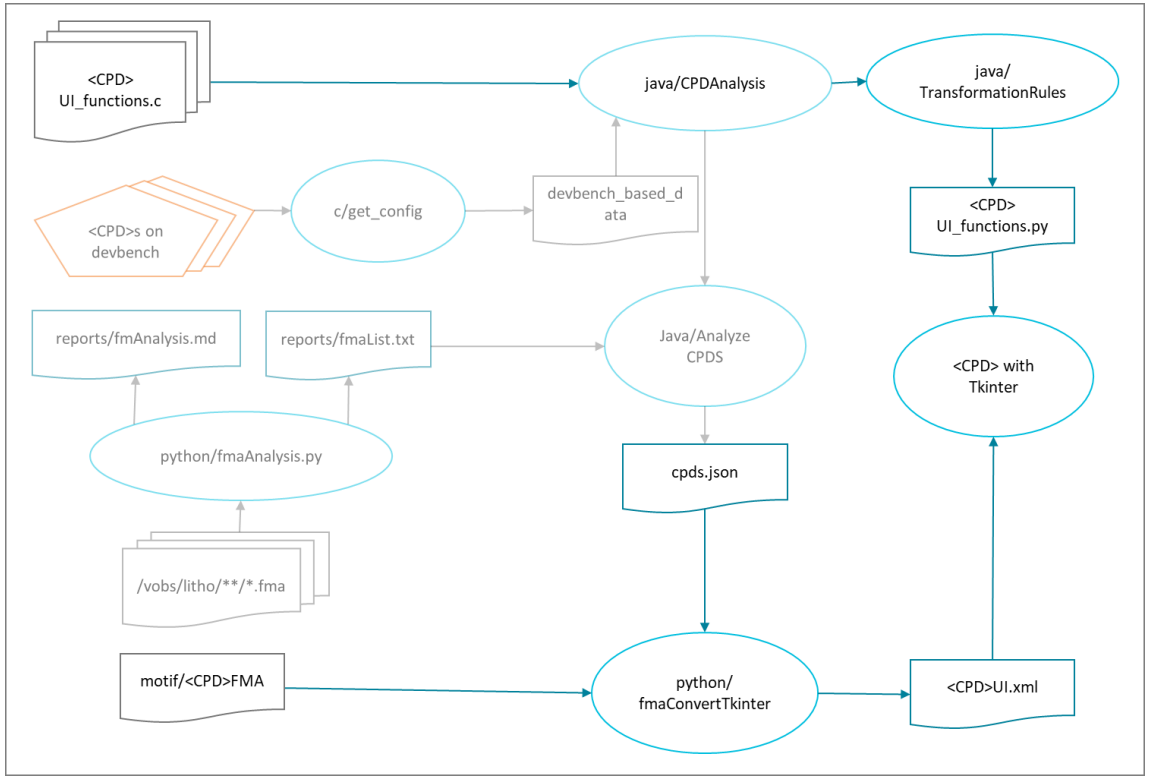
TRANSPILATION

- Exploit structure in code
  - Business logic – not migrated
    - Call C function from Python
  - UI layout
    - Dedicated script written to convert resource file from Motif's FrameMaker to Tkinter format
  - UI logic from C to Python
    - Not performance critical
    - Not all C constructs used, e.g., no pointers
    - Some challenging C constructs used, e.g., macros



TRANSPILATION

- Exploit structure in code
  - Same organization
    - Folders
    - Files
    - Functions
  - Readability and maintainability
    - High level: Same structure
    - Low level: Idiomatic Python

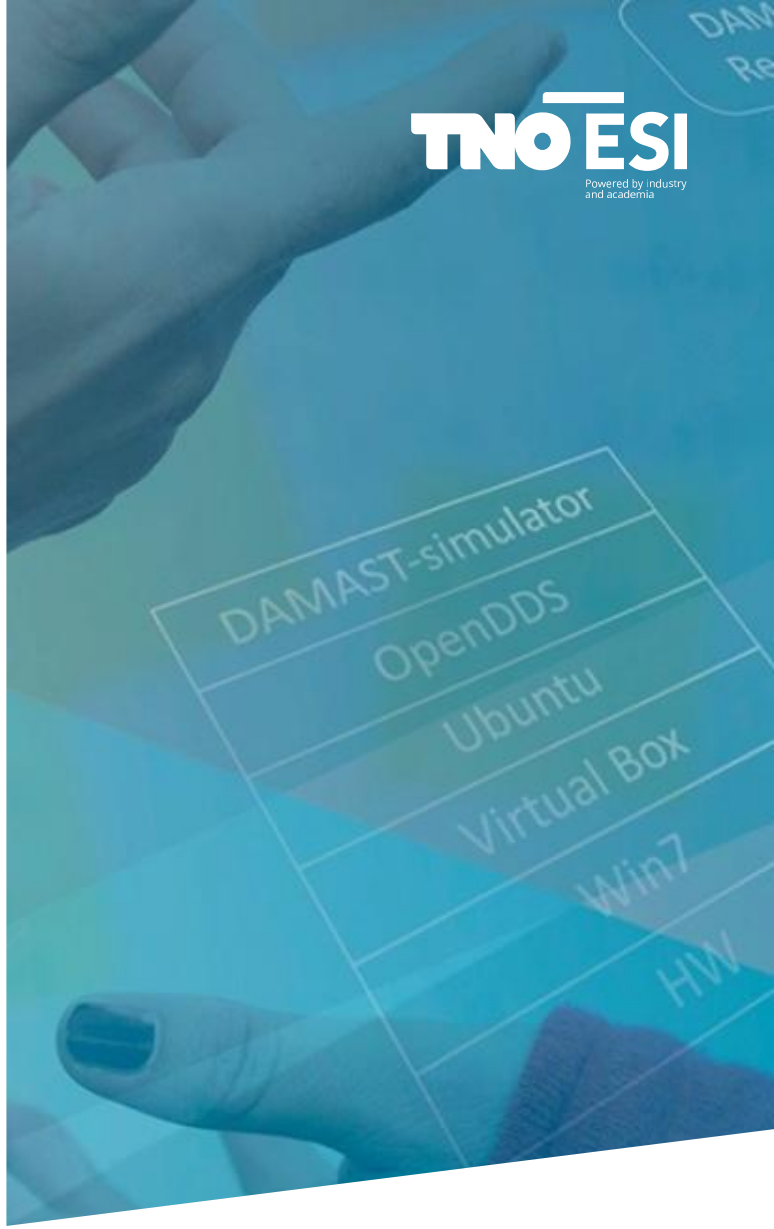


RESEARCH QUESTION FOR USE CASE

Estimation for manual transpilation

100 man-years

Can automation reduce the effort?



3

LEARN BY EXAMPLE

LEARN BY EXAMPLE

C code

```
…
int a[N];
for (int i = 0; i < N; i++)
{
  a[i] = i * i;
}
…
```

LEARN BY EXAMPLE

C code

```
…
int a[N];
for (int i = 0; i < N; i++)
{
  a[i] = i * i;
}
…
```

Rule

```
$type $var[$size];
for (int $i = 0; $i < $size; $i++)
{
  $var[$i] = $expr;
}
→
$var = [$expr for $i in range($size)]
```

LEARN BY EXAMPLE

C code

```
…
int a[N];
for (int i = 0; i < N; i++)
{
  a[i] = i * i;
}
…
```

Rule

```
$type $var[$size];
for (int $i = 0; $i < $size; $i++)
{
  $var[$i] = $expr;
}
→
$var = [$expr for $i in range($size)]
```

LEARN BY EXAMPLE – PLACEHOLDER & RULES

C code of $expr	Rule – Literal
i * i	$lhs * $rhs → coverflow($lhs * $rhs)

Rule – Idiomatic

$lhs * $rhs → $lhs * $rhs

LEARN BY EXAMPLE

C code

```
…
int a[N];
for (int i = 0; i < N; i++)
{
  a[i] = i * i;
}
…
```

Rule

```
$type $var[$size];
for (int $i = 0; $i < $size; $i++)
{
  $var[$i] = $expr;
}
→
$var = [$expr for $i in range($size)]
```

Python literal code

```
…
a = [coverflow(i * i) for i in range(N)]
…
```

Python idiomatic code

```
…
a = [i * i for i in range(N)]
…
```

4

RENAISSANCE APPROACH FOR TRANSPILATION

REQUIREMENT: MAINTAINABLE CODE

- Determine whether C-language- and compiler-specific behaviour occurs
  - Transpile to non-idiomatic Python code that throws exception whenever specific behaviour occurs

- Test non-idiomatic Python code extensively

- No exception occurred → Transpile to idiomatic Python code
- Exception occurred → Determine best way forward
  - Adept C-code
  - Non-idiomatic transpilation to Python
  - Reimplement in Python
  - …

REQUIREMENT: MAINTAINABLE CODE

- Ensure ASML can easily maintain generated Python code

  i++ → (i, i := i + 1)[0]

- Idiomatic Python instead of literal transpilation
  - Acceptable for both compiler / interpreter and human developers
  - Prevent that expertise in C, Python, and on the actual transpilation is needed to maintain the code
  - Is literal transpilation even possible for undefined behaviour?

- Exploit regularities, conventions, and patterns in the original C code

  i + 1 > 0 ⇔ i >= 0

  Does overflow occur for i == INT_MAX ?

- Keep particularities of the C-language out of the generated Python code
  - Keep functional behaviour and remove history of development
  - Awareness of C-language- and compiler-specific behaviour

REQUIREMENT: EFFICIENCY

- Count usage of C constructs and UI widgets
  - Frequently occurring → Automatic transpilation
  - Rarely occurring → Manual transpilation
  - Not occurring → No effort needed

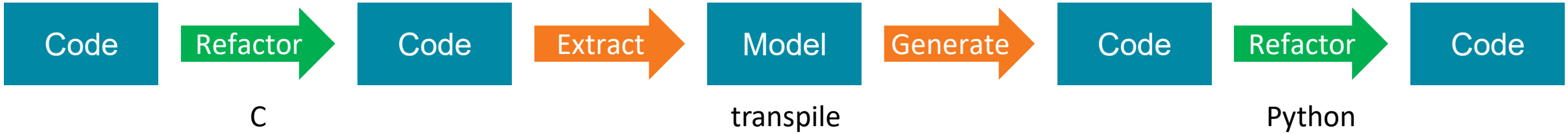
- Clear interface: What must be done manually?
  - For both human / developers and computer / version control system

REQUIREMENT: EFFICIENCY

- Generate analysable code, also for the manual parts
  - In Python, use “pass” for statements and “None” for expressions
  - Use comments to specify what must be done

- Communicate what is exactly done
  - Remarks that help testers and reviewers

PRINCIPLE: PREPARE, TRANSPILE, FINALIZE



- From human readable, maintainable C code to C code prepared from transpilation to Python
- Transpilation
  - described using rewrite rules
  - based on AST
- Exploit existing tooling for Python as much as possible to simplify transpilation
  - Formatters, pretty printers, beautifiers, (expression) simplifiers, linters, …

PRINCIPLE: PREPARE, TRANSPILE, FINALIZE

C

Increase regularity of code

- Standardize if statements

  if (cond) return 0;

  if (cond) **{** return 0; **}**

- Standardize for loops

  for (i=0; i < 10; **i++**) { f(i); }

  for (i=0; i < 10; **++i**) { f(i); }

- Standardize if statements

  if (cond) **{** return 0; **}**

  if (cond) **{** return 0; **}**

- Standardize for loops

  for (i=0; i < 10; **i++**) { f(i); }

  for (i=0; i < 10; **i++**) { f(i); }

PRINCIPLE: PREPARE, TRANSPILE, FINALIZE

C

Increase regularity of code

- Separate declaration and initialization in C

```
int sqr[10];
int cube[20];

for (int i = 0; i < 10; i++) { sqr[i] = i*i; }
for (int i = 0; i < 20; i++) { cube[i] = i*i*i; }
```

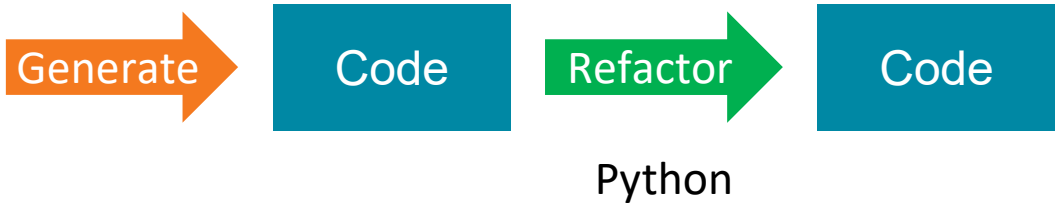
- Desired idiomatic Python

```
sqr: list[int] = [ i*i for i in range(10) ]
cube: list[int] = [ i*i*i for i in range(20) ]
```

- Prepare for pattern matching

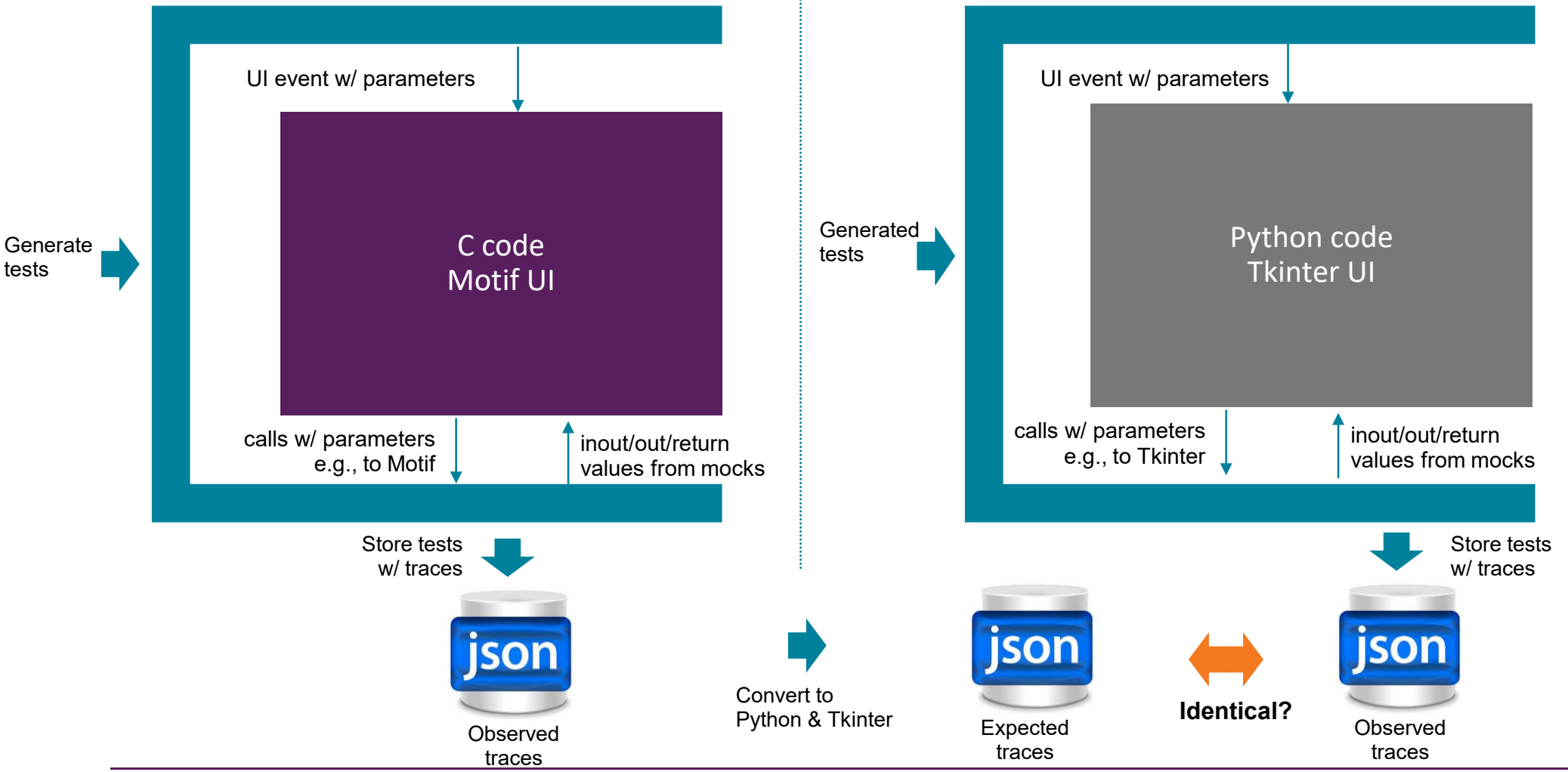
```
int sqr[10];
for (int i = 0; i < 10; i++) { sqr[i] = i*i; }

int cube[20];
for (int i = 0; i < 20; i++) { cube[i] = i*i*i; }
```

PRINCIPLE: PREPARE, TRANSPILE, FINALIZE



- Example of investigation
  - Precedence of operators differs between C and Python
  - Simplest solution – just add brackets everywhere and let tool remove the redundant ones
  - Unfortunately, no tool for Python exists that removes redundant brackets
  - So transpilation had to handle difference in precedence

PRINCIPLE: KEEP FUNCTIONAL BEHAVIOR



REJUVENATION METHODOLOGY

Analysis

Automation

Validation

REJUVENATION METHODOLOGY

Analysis

- Code base analysis
  - Frequency of
    - UI widgets
    - Language constructs
    - Patterns
- Languages analysis
  - Commonalities and differences
  - Contexts that expose differences

REJUVENATION METHODOLOGY

Automation

- Rules
  - Find and replace – sequence of AST nodes
  - Placeholders – single and multi
- Rule engine
  - Applies rules in given order
  - Recursive on placeholders
  - Catch all – manual action required
- Rule organization
  - Simplify handling different scenarios
  - Simplify iteration

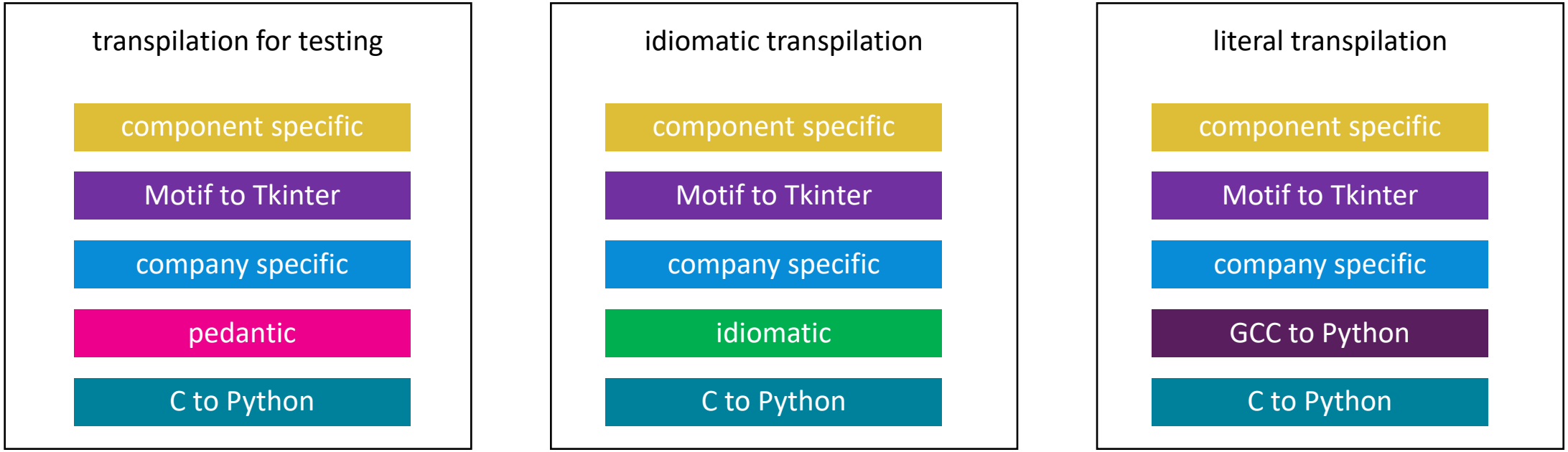
REJUVENATION METHODOLOGY

Validation

- Transpilation rules – Unit tests
  - Compilable
  - Adhere to style
  - Correct
- Functional regression – End to end tests
  - Context of usage
    - Which differences can be ignored?
  - Correct transpilation
    - Equivalent traces on same test vectors

RENAISSANCE APPROACH: RULE ORGANIZATION

- Separate lists provides structure
- Enables different transpilations and our way-of-working



RENAISSANCE APPROACH: RULE ORGANIZATION

- Separate lists provides structure
- Enables different transpilations and our way-of-working

```
z += x;
```

transpilation for testing

```
def except_overflow(value):
    if not(-2**31 <= value < 2**31):
        raise Exception("Integer overflow")
    return value


z = except_overflow(z + x)
```

idiomatic transpilation

```
z += x
```

literal transpilation

```
def emulate_overflow(value):
    return (value + 2**31) % 2**32 - 2**31


z = emulate_overflow(z + x)
```

ACTIVE WIDGET: LITERAL

```
int nrofWidgets = 8;

int toWidgetId(int id) {
    return (nrofWidgets + id) % nrofWidgets;
}

int next (int widgetId) {
    return toWidgetId(widgetId + 1);
}

int previous (int widgetId) {
    return toWidgetId(widgetId - 1);
}
```

```
def cmod(numerator, denominator):
    pmod = numerator % denominator
    if pmod == 0:
        return pmod
    elif numerator * denominator < 0:
        return pmod - denominator
    else:
        return pmod

def coverflow(value):
    return (value + 2**31) % 2**32 - 2**31

nrofWidgets = 8

def toWidgetId(id):
    return cmod((coverflow(nrofWidgets + id)),
                nrofWidgets)

def next(widgetId):
    return toWidgetId(coverflow(widgetId + 1))

def previous(widgetId):
    return toWidgetId(coverflow(widgetId - 1))
```

ACTIVE WIDGET: CONTEXT OF USAGE

```
int nrofWidgets = 8;

int toWidgetId(int id) {
    return (nrofWidgets + id) % nrofWidgets;
}

int next (int widgetId) {
    return toWidgetId(widgetId + 1);
}

int previous (int widgetId) {
    return toWidgetId(widgetId - 1);
}
```

```
def mod(numerator, denominator):
    if numerator < 0 or denominator < 0:
        raise Exception("Negative argument");
    return numerator % denominator

def overflow(value):
    if not(-2**31 <= value < 2**31):
        raise Exception("Integer overflow")
    return value

nrofWidgets = 8

def toWidgetId(id):
    return mod((overflow(nrofWidgets + id)),
               nrofWidgets)

def next(widgetId):
    return toWidgetId(overflow(widgetId + 1))

def previous(widgetId):
    return toWidgetId(overflow(widgetId - 1))
```

ACTIVE WIDGET: USING NATIVE OPERATORS

```
int nrofWidgets = 8;

int toWidgetId(int id) {
    return (nrofWidgets + id) % nrofWidgets;
}

int next (int widgetId) {
    return toWidgetId(widgetId + 1);
}

int previous (int widgetId) {
    return toWidgetId(widgetId - 1);
}
```

```
nrofWidgets = 8

def toWidgetId(id):
    return (nrofWidgets + id) % nrofWidgets

def next(widgetId):
    return toWidgetId(widgetId + 1)

def previous(widgetId):
    return toWidgetId(widgetId - 1)
```

ACTIVE WIDGET: MAINTAINABLE PYTHON CODE

```c
int nrofWidgets = 8;

int toWidgetId(int id) {
    return (nrofWidgets + id) % nrofWidgets;
}

int next (int widgetId) {
    return toWidgetId(widgetId + 1);
}

int previous (int widgetId) {
    return toWidgetId(widgetId - 1);
}
```

```python
nrofWidgets = 8

def toWidgetId(id):
    return id % nrofWidgets

def next(widgetId):
    return toWidgetId(widgetId + 1)

def previous(widgetId):
    return toWidgetId(widgetId - 1)
```

5

RESULTS OF USE CASE

AREAS OF ATTENTION FOR TRANSPILATION

Areas of Attention	Examples	C	Python
difference in signatures of operators and functions	division (/) and & or operator	division: int × int → int e.g., 1/5 → 0 and: int × int → bool e.g., 1 && 13 → 1	division: int × int → float e.g., 1/5 → 0.2 and: int × int → int e.g., 1 and 13 → 13
difference in properties of operators	associativity, chainable, and evaluation (lazy/eager)	no chaining e.g., -1 < 0 < 1 → False	chaining e.g., -1 < 0 < 1 → True
difference in the relative precedence of operators	equality (==) and bit-wise and (&)	equality has precedence over bit-wise and 1 & 3 == 1 → False	bit-wise and has precedence over equality 1 & 3 == 1 → True
operators and functions whose behavior is “under discussion”	mod & div operator division (0/0) power (0^0)	same sign as dividend e.g., -1 % 3 → -1 0 / 0 → 0	same sign as divisor e.g., -1 % 3 → 2 0 / 0 → division by zero error
difference in data types	array int	array has no length information 2147483647 + 1 → -2147483648	array has length information 2147483647 + 1 → 2147483648
difference in invariants	range of data types modulo	-2147483648 <= i → True (N + i) % N ≠ i % N	-2147483648 <= i → dependent on i (N + i) % N ⇔ i % N

BENEFIT OF RENAISSANCE APPROACH

- ASML selected two components to get insight in Renaissance approach and to estimate its benefits
- ASML supported TNO ESI to transpile components
- ASML estimated benefit of 70 %

6

COMPARISON

COMPARISON WITH AI (2024)

- Brief experiment

- AI did not handle any of the problematic areas we found correctly

- Similar effort – provide rewrite rules to AI
- Non-deterministic output – output still to be checked
- So, our methodology preferred over AI

- Whether that is still the case?

7

SUMMARY

SUMMARY USE CASE

- Not a C to Python transpiler
  - Only the ASML code base
  - Only the UI logic
- Effective use of automation
  - Automation must be worthwhile
  - Only frequently occurring patterns
  - Prepare, transpile, and finalize
- Estimation for manual transpilation
  - 100 man-years
- Can automation reduce the effort?
  - Estimation of 70% reduction

QUESTIONS?

- Contact me at pierre.vandelaar@tno.nl
- Article at https://doi.org/10.1109/SANER67736.2026.00031