The EKEKO/X Program Transformation Tool

Coen De Roover[†*] and Katsuro Inoue[†]
[†]Software Engineering Laboratory, Osaka University, Osaka, Japan
[*]Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
Email: cderoove@vub.ac.be, inoue@ist.osaka-u.ac.jp

***Abstract*—Developers often need to perform repetitive changes to source code. For instance, to repair several instances of a bug or to update all clients of a library to a newer version. Manually performing such changes is laborious and error-prone. Program transformation tools enable automating changes, but specifying changes as a program transformation requires significant expertise. Code templates are often touted as a remedy, yet have never been endorsed wholeheartedly. Their use is mostly limited to expressing the syntactic characteristics of the intended change subjects. Less familiar means have to be resorted to for expressing their structural, control flow, and data flow characteristics. In this tool paper, we introduce a decidedly template-driven program transformation tool called EKEKO/X. Its specifications feature templates for specifying all of the aforementioned characteristics of its subjects. To this end, developers can associate different directives with individual components of a template. Each matching directive imposes particular constraints on the matches for the component it is associated with. Rewriting directives, on the other hand, determine how each match should be changed. We develop EKEKO/X from the ground up, starting from its applicative logic meta-programming foundation. We highlight the key choices in this implementation and demonstrate its use through two example program transformations.**

I. INTRODUCTION

Manually performing similar changes to dispersed locations in the source code can be laborious and error-prone. Some of the required changes may be overlooked, and some unwarranted changes may be performed. Program transformation tools enable automating changes, but specifying changes as a program transformation requires significant expertise.

In this paper, we introduce EKEKO/X as a new Eclipse plugin for transforming Java programs.[1] As in most approaches to program transformation, its specifications consist of a left-hand side (LHS) and a right-hand side (RHS) component. The left-hand side component identifies the subjects of the transformation, while the right-hand side component defines how each identified subject should be changed. EKEKO/X specifications are decidedly template-driven. On the right-hand side, code templates are used as intuitive short-hands for complex code generation expressions. On the left-hand side, code templates are used to specify the characteristics of the intended transformation subjects. These subjects correspond to the matches in the source code for each code template.

II. AN APPLICATIVE LOGIC FOUNDATION

EKEKO/X owes its peculiar name to the Clojure library on top of which it is implemented. EKEKO [1] enables querying and manipulating an Eclipse workspace using logic queries that are embedded in functional expressions. To this end, it provides a comprehensive collection of both declarative predicates and functions that abstract over the low-level APIs of the Eclipse platform. Recent applications of EKEKO include detecting suspicious aspect-oriented code [2] and detecting fine-grained evolutions of versioned code [3]. In this section, we demonstrate how EKEKO also lends itself to providing the foundation for a program transformation tool.

The following listing depicts the typical implementation of a straightforward program transformation in EKEKO. The transformation is to wrap `int`-valued arguments to invocations of a method `setAge` in an explicit `Integer` object. For instance, `this.setAge(age++)` should be changed into `this.setAge(new Integer(age++))` provided that `age++` is an `int`-valued expression. We will build EKEKO/X from the ground up illustrating the shortcomings and advantages of its intermediate stages using this running example.

```
(doseq  [[subject &rest]                                              1
  (ekeko [?subject ?name ?inv]                                        2
      (ast :MethodInvocation ?inv)                                    3
      (has :name ?inv ?name)                                          4
      (name|simple-string ?name "setAge")                             5
      (child :arguments ?inv ?subject)                                6
      (ast|expression-type|primitive ?subject "int"))]                7
 (replace-node                                                        8
   subject                                                            9
   (newast :ClassInstanceCreation                                     10
           :arguments (list subject)                                  11
           :type (newast :SimpleType                                  12
                         :name                                        13
                         (newast :SimpleName                          14
                                 :identifier "Integer"))))) 15
```

A. Identifying Transformation Subjects

The `ekeko` special form on line 2 launches a logic query that identifies the subjects of this transformation. It takes a vector of meta-variables, each denoted by a starting question mark, followed by a sequence of logic goals. Solutions to the query consist of the different bindings for its meta-variables such that all logic goals succeed. Internally, the EKEKO engine performs an exploration of all possible results, using backtracking to yield the different bindings for the meta-variables in the query. Evaluated against the above example, the query's solutions would include a 3-tuple `[age++ setAge this.setAge(age++)]`.

The goals of the query bind `?inv` to a `MethodInvocation` from the program under transformation (line 3) such that `?name` is the value of its `name` property (line 4) and `?subject` is

[1]The implementation of EKEKO/X and a video demonstration is available at https://github.com/cderoove/damp.ekeko.snippets.

an element of its list-valued `arguments` property (line 6). To this end, each goal uses a logic predicate from the EKEKO library that reifies a syntactic relation about the program under transformation. Binary predicate `ast/2` reifies the relation of all AST nodes of a particular type, denoted by the capitalized name of the node's class. Ternary predicate `has/3` reifies the relation between an AST node and the value of one of its properties, denoted by the uncapitalized name of the property.

EKEKO provides similar logic predicates that reify structural, control flow and data flow relations. For instance, binary predicate `ast|expression-type|primitive/2` is used on line 7 to further restrict `?subject` to an int-valued expression.[2] The EKEKO library is accompanied by an Eclipse plugin that maintains each of these relations by continuously listening for developer changes. As a result, the information about the program under transformation is always up-to-date.

B. Changing Transformation Subjects

A Clojure expression of the form `(doseq [<el> <exp_coll>] <body_exp>)` surrounds the logic query. It evaluates `<body_exp>` for every element `<el>` in the collection `<exp_coll>`. In our case, `<exp_coll>` corresponds to the solutions for the logic query (lines 2–7); a collection of 3-tuples of which the first attribute is to be changed by the transformation. Within `<body_exp>` (lines 8–15), the Clojure destructuring form `[subject &rest]` (line 1) binds Clojure variable `subject` to this attribute of each solution tuple. This enables changing the subject using functions `replace-node` and `newast` provided by the EKEKO library.

Like the aforementioned predicates, these functions take the names of classes that are to be instantiated and the names of properties that are to be assigned. Together, they provide a functional interface to the Eclipse rewriting API. Note that these functions stage their changes in a so-called `ASTRewrite` until the user calls `apply-and-reset-rewrites`. It is only at that point that the source code of the program is actually changed. In turn, these changes will trigger the EKEKO plugin to update its relations for subsequent logic queries.

C. The Need for Expressive Code Templates

Expertise is required to implement a program transformation using the predicates and functions provided by the EKEKO library. While logic and functional programming can be effective for specifying the characteristics of and changes to the subjects of a transformation, it is far from intuitive.

Some predicates and functions even expose implementation details to the user. In our running example, this was the case for the predicates `ast/2`, `has/3`, and `child/3`. They require an intricate familiarity with the abstract grammar used by the Eclipse JDT for AST nodes. The `:MethodInvocation` node bound to `?inv` on line 12, for instance, does not have a property named `:identifier`. The same goes for the functions that provide an interface to the rewrite API. For instance, only a `:SimpleName` node can be used as the value for the `:name` property of the newly created `:SimpleType` node on line 12.

To remedy this shortcoming, a great deal of query and transformation tools incorporate concrete syntax of the program under investigation [4], [5], [6], [7], [8], [9], [10], [11]. However, code templates are seldomly used to specify non-syntactic characteristics of source code. For instance, to specify that an invocation should call a particular method or that a variable should refer to a particular field. The query tool SOUL [12], the predecessor to EKEKO, is a notable exception. It features a very lenient matching strategy such that code templates also match implementation variants of their structural, control flow and data flow characteristics. To prevent unwarranted changes, however, a transformation tool requires a means for developers to exert more control over the way templates are matched.

III. TOWARDS TEMPLATE-DRIVEN TRANSFORMATION

EKEKO/X extends EKEKO with a logic goal `(match <ast> <template>)` that is satisfied for every `<ast>` from the program under transformation that matches the given code `<template>`. Here, `<template>`[3] is concrete syntax (*e.g.,* a snippet of Java code) in which meta-variables substitute for unknowns. Using a code template, our running example can be re-specified as follows. Here, `change-int-to-integer` is a function of which the body corresponds to lines 8–15 of the original Clojure expression:

```
(doseq  [[subject &rest]                                    1
  (ekeko [?subject ?inv ?exp]                               2
    (match ?inv "?exp.setAge(?subject)"))]                  3
  (change-int-to-integer subject))                          4
```

In solutions to the query on lines 2–3, `?inv` will be bound to an invocation of a method `setAge` of which `?exp` is the receiver and `?subject` is the single argument. They would, for instance, include the 3-tuple `[age++ this this.setAge(age++)]`.

A. Matching Directives for LHS Templates

Whether an AST node matches a code template depends on particular matching strategy. EKEKO/X enables specifying a matching strategy as a combination of separate directives.

The default matching strategy, for instance, establishes a tree homomorphism between a node N_p of the program (with parent node P_p) and a node N_t of the template (with parent node P_t). It consists out of directives `child/0` and `match/0`. Directive `child/0` is satisfied when P_p matches P_t, and N_p is the value of the corresponding node-valued property of P_p.[4] Directive `match/0` is satisfied when both nodes are of the same type, and their non-node valued properties have the same value. For every template node, `match/0` filters out unwanted matches from the candidates retrieved from the program's source code by `child/0`.

Additional matching directives `<directive-name>` or `(<directive-name> <arg1> ...<argn>)` can be associated

[2]The names of predicates that reify an n-ary relation consist of n components separated by a -, each describing an element of the relation. Vertical bars | separate words within the description of a single component.

[3]In this section, we render code templates as code between string delimiters. Section IV discusses how the actual rendering is malleable.

[4]Any node of the same type is a candidate match for the template's root.

with individual components of a template using a [<component>]@[<directive$_1$> ...<directive$_n$>] syntax:

```
(doseq  [[subject &rest]
  (ekeko [?subject ?inv ?exp]
    (match ?inv
          "[?exp]@[orimplicit].setAge(?subject)"))]
  (change-int-to-integer subject))
```

Here, we have added an orimplicit/0 directive that overrides the default match/0 strategy for invocation receivers. Indeed, the dot in the concrete syntax "?receiver.setAge(?subject)" inherently disallows matches without a receiver. The newly-added constraint ensures that a method invocation setAge(5) with an implicit this receiver will match our code template.

Table I lists some representative matching directives covering the different kinds of source code characteristics. We will discuss examples of their use in Section V.

B. Rewriting Directives for RHS Templates

Similar ideas can be transposed to the right-hand side of a transformation specification. Here, templates serve as intuitive short-hands for expressions that generate code. To this end, EKEKO/X provides a Clojure function (instantiate <SUBST> <RHS-1> ... <RHS-n>) that takes a substitution (*i.e.,* a map from meta-variables to their binding) and a variable amount of right-hand side templates. It instantiates these templates by invoking EKEKO's functional interface to the Eclipse ASTRewrite API in a recursive descent through each template. This results in calls similar to those on lines 10–15 of the plain EKEKO specification in Section II.

Instantiation of meta-variables within a template is delayed until the other elements of all templates have been instantiated. This is because the binding for meta-variables within an RHS template either stem from the match for a LHS template (*i.e.,* the initial substitution argument to the function), or from the code that is to be instantiated for another RHS template. As such, a meta-variable in the first RHS template might receive its binding from the instantiation of the last RHS template.

To specify where the generated code should be inserted, rewriting directives can be associated with the root of a RHS template. The syntax is the same as for the matching directives on the LHS. In general, rewriting directives take a single meta-variable as their argument. Examples include (replace ?var) or (add-element ?lstvar) which respectively result in their argument being replaced by or being extended with the instantiation of the template they are associated with. A single matching directive, (equals ?var), is supported within RHS templates to bind ?var to a node within the instantiated code. Section V discusses examples of its use.

C. Template-Driven Transformation Specifications

A Clojure macro (ekeko/x <LHS-1> ... <LHS-n> => <RHS-1> ... <RHS-n>) renders the final specification for our running example more succinct:

```
(ekeko/x
   [...]@[orimplicit].setAge([?s]@[(type|sname "int")])
   =>
   [new Integer(?s)]@[(replace ?s)])
```

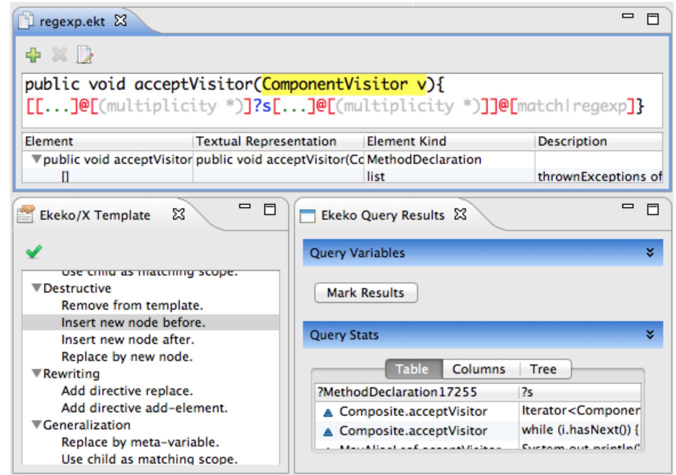

Fig. 1: Our editor on a LHS template and its matches.

This macro performs the changes specified by its second argument <RHS> to all matches for its first argument <LHS>. To this end, it merely has to expand into the familiar doseq-expression with the appropriate meta-variable declarations.

Note that apart from meta-variables, some non-Java syntax is allowed within LHS templates. Above, a wildcard ... substitutes for the actual receiver of the method invocation. Such a wildcard matches any node from the base program, eliminating an otherwise unused meta-variable.

IV. IMPLEMENTATION HIGHLIGHTS OF THE TOOL

Before illustrating the use of EKEKO/X, we briefly highlight and motivate some key choices in its implementation.

A. Implemented as an Extension to EKEKO

First, as evidenced by the intermediate stages of our running example, we opted to implement EKEKO/X by extending EKEKO rather than merely building on top of it. As a result, functional and logic programming can be resorted to wherever the default EKEKO/X semantics fall short. This also facilitates implementing alternative template matching and transformation application strategies.

Both the match goal (cf. Section III-A) and the instantiate function (cf. Section III-B) of EKEKO/X are expanded at compile-time into invocations of the logic predicates and functions provided by EKEKO. The matching and rewriting directives within the respective templates control the expansion. To this end, we perform a recursive descent through a template and ask the directives associated with each encountered template element to expand themselves in the context of the element and the template. Clojure's support for manipulating symbolic expressions (*i.e.,* its syntax-quote, unquote and splicing constructs) greatly facilitates such code generation tasks. In fact, new directives can be added easily.

B. Implemented without a Template Parser

Second, to speed up prototyping, we wanted to avoid committing to a particular syntax for templates early on. We

<table>
<tr><th>Directive</th><th>Template element</th><th>Constraints on match for template element.</th></tr>
<tr><td>child, child+, child*</td><td>Any</td><td>Match is the corresponding child of the parent match, nested within that child (+), or either (*).</td></tr>
<tr><td>(equals ?var)</td><td>Any</td><td>Match unifies with the given meta-variable. Used to expose the match for the template element.</td></tr>
<tr><td>match</td><td>Any</td><td>Type of match and its non-node valued properties are the same.</td></tr>
<tr><td>orimplicit</td><td>Invocation receiver</td><td>As above, except that implicit this-receivers also match.</td></tr>
<tr><td>orsimple</td><td>Qualified name</td><td>As above, but unqualified package or type names that resolve to the name in the template match.</td></tr>
<tr><td>match|set</td><td>List</td><td>List of which the elements match a set corresponding to the template element.</td></tr>
<tr><td>match|regexp</td><td>List</td><td>List of which the elements match a regular expression corresponding to the template element.</td></tr>
<tr><td>match|regexp-path</td><td>Statement list</td><td>List of which the elements match a path through the control flow graph for the template element.</td></tr>
<tr><td>(multiplicity +/*/n)</td><td>Regexp list element</td><td>Multiplicity of matches within a regexp-matched list: at least one (+), 0 or more (*), exactly n.</td></tr>
<tr><td>(type ?type)
(type|sname <string>),
(type|qname <string>)</td><td>Type or variable declaration or reference, expression</td><td>Match resolves to or declares the type, declares or references a variable of the type, or is an expression of the given type ?type or its simple or qualified name <string>.</td></tr>
<tr><td>(subtype+ ?type), ...</td><td>As above</td><td>Same as above, except that the match is required to resolve to a transitive subtype of the argument.</td></tr>
<tr><td>(subtype* ?type), ...</td><td>As above</td><td>Same as above, except that the match is required to resolve to a reflexive transitive subtype.</td></tr>
<tr><td>(refers-to ?var)</td><td>Expression</td><td>Match lexically refers to a local variable, parameter or field denoted by the argument.</td></tr>
<tr><td>(referred-by ?exp)</td><td>Variable</td><td>Match declares a local variable, parameter or field lexically reffered to by the expression denoted by its argument.</td></tr>
<tr><td>(invokes ?method),
(invokes|qname <string>)</td><td>Invocation</td><td>Match is a (super/constructor) invocation that invokes the given argument according to the declared static type of its receiver.</td></tr>
<tr><td>(invoked-by ?inv)</td><td>Method, constructor</td><td>Inverse of the above.</td></tr>
<tr><td>(may-alias ?exporvar)</td><td>Expression, variable</td><td>Match may alias the given expression or variable, according to an intra-procedural analysis.</td></tr>
</table>

TABLE I: Representative matching directives, the template elements they can be applied to, and the constraints they impose.

forwent developing a template parser altogether. Instead, we implemented code templates as a data structure that wraps a regular abstract syntax tree provided by the Eclipse JDT parser. This data structure maps each template element to a hidden match meta-variable and to a list of matching or rewriting directives. Expanding the latter, at compile-time, results in an expression that will, at run-time, bind the former to the match or the instantiation for the template element.

Of course, transformation specifications still need to be created, edited and persisted. We therefore developed an Eclipse plugin that enables users to create a LHS or RHS template from a selected code snippet. Matching and rewriting directives can be associated with the elements of the resulting specification which, in turn, is pretty-printed to the particular concrete syntax used throughout this paper. The default matching directives are not printed. Figure 1 depicts the transformation editor on a LHS template `"regexp.ekt"` that uses regular expression matching (cf. Table I) to bind `?s` to any statement within the body of an `acceptVisitor` method.

The editor at the top of the screenshot highlights the currently selected template element in yellow. To change this element or its matching directives, users can select and apply an operator from the bottom-left view. To test the resulting template, users can match it against the program under transformation at any time. The bottom-right view depicts the current matches. Once the users are satisfied that these matches are correct, they can have the transformation applied according to their RHS templates. We do not yet support a preview of the actual changes, but our use of the `ASTRewrite` API enables doing so in the future using the conventional Eclipse dialogs.

Persisting transformation specifications requires persisting the AST nodes that underlie their LHS and RHS templates, something the Eclipse JDT does not support. However, Clojure supports extending its persistency protocol to existing Java hierarchies. Doing so, we are able to read and write specifications from the Clojure run-time, enabling the functionality demonstrated in the previous section:

```
(let [template (slurp "regexp.ekt")]                    1
  (ekeko [?method] (match ?method template)))           2
```

Note that our plugin supports editing and applying all transformations supported by the `ekeko/x` macro of the previous section, without exposing developers to Clojure. To this end, its graphical user interface calls back into Clojure.

V. Example Transformations using Ekeko/X

We illustrate Ekeko/X using examples of repetitive changes that the first author had to perform while contributing to a change-centric software meta-model [13].

A. Example: Adding Type Parameters

Figure 2 illustrates the particular changes that need to be performed for the first example. The raw `Identifier` type of those fields that carry an `@EntityProperty` annotation, is to receive a type parameter that corresponds to the annotation's `value` key. We will develop the Ekeko/X transformation that automates these changes in an incremental manner.

Figure 3 depicts the initial specification for this example. Lines 1–2 correspond to its LHS, while line 4 corresponds to its RHS. The single template on the LHS matches the field declarations of which the type is to change. Such declarations have the appropriate annotation among their list of annotations. Line 1 therefore uses directive `match|set` to allow matches with additional annotations in any order. Meta-variable `?annoType` corresponds to the type parameter that is to be used for the new type of the field declaration. Meta-variable `?fieldType` corresponds to the old type that is to be replaced. It is bound through matching directive `equals`. The RHS of the specification instantiates its single template to a new parameterized type and replaces the `?fieldType` with it through rewriting directive `replace`.

Figure 4 depicts a more refined version of this specification that also changes the getter and setter methods for the field accordingly. To this end, it groups these declarations together

```
//Before changes:                                         1
@EntityProperty(value = SimpleName.class)                 2
private Identifier label;                                 3
//After changes:                                          4
@EntityProperty(value = SimpleName.class)                 5
private Identifier<SimpleName> label;                     6
```

Fig. 2: Example of changes for the type parameter case.

```
[@EntityProperty(value=?annoType.class)]@[match|set]      1
[Identifier]@[(equals ?fieldType)] ...;                   2
=>                                                        3
[Identifier<?annoType>]@[(replace ?fieldType)]            4
```

Fig. 3: Initial specification for the type parameter case.

in the body declarations of class. The `match|set` directive is used once more to ensure that matches can feature additional declarations in any order. A single class can therefore feature as the match for the LHS template multiple times, once for each 3-tuple of a field and its accessor methods. Lines 5 and 9 rely on the `refers-to` directive to ensure that the getter and setter actually operate upon the `?field` that matches line 3. Lines 4 and 7 respectively extract their return and parameter type, which receive a type parameter on lines 14 and 15. The deep matching directives `child*` on lines 3, 4, 7 ensure that the types of the form `List<Identifier>` in the program also match the `Identifier` types in the template.

B. Example: Generating a Visitor for a Class Hierarchy

Figure 5 illustrates the repetitive changes that need to be performed when implementing a Visitor for the same class hierarchy. Every class in the `ASTNode` hierarchy is to receive a `acceptVisitor` method that double dispatches to a corresponding `visit<Class>` method that is to be declared in an existing, but empty `IASTVisitor` interface. Note that this requires changing the import declarations of the compilation units in which these classes reside.

Figure 6 depicts the EKEKO/X specification that automates these changes. Both its LHS and RHS consist of multiple templates. The template on lines 1–3 identifies the classes in our hierarchy and binds them and their list of body declarations to `?visited` and `?bodyVisited` respectively. To this end, the

```
... class ... {                                           1
[ @...(value=?annoType.class)                             2
  private [Identifier]@[child* (equals ?fieldType)] ?field; 3

  public [Identifier]@[child* (equals ?returnType)] ...(){ 4
    return [...]@[(refers-to ?field)];                    5
  }                                                       6

  public void ...([Identifier]@[child* (equals ?paramType)] 7
                  ?param){                                8
    [...]@[(refers-to ?field)]=[...]@[(refers-to ?param)]; 9
  }                                                       10
]@[match|set]}                                            11

=>                                                        12

[Identifier<?annoType>]@[(replace ?fieldType)]            13
[Identifier<?annoType>]@[(replace ?returnType)]           14
[Identifier<?annoType>]@[(replace ?paramType)]            15
```

Fig. 4: Final specification for the type parameter case.

```
//Visitor compilation unit after changes:                 1
import be.ac.chaq.model.ast.java.visitor.IASTVisitor;     2
public class BreakStatement extends Statement {           3
  public void acceptVisitor(IASTVisitor visitor){         4
    visitor.visitBreakStatement(this);                    5
  } //...                                                 6
}                                                         7
//Visited compilation units after changes:                8
import be.ac.chaq.model.ast.java.BreakStatement;          9
public interface IASTVisitor {                            10
  public void visitBreakStatement(BreakStatement o); //... 11
}                                                         12
```

Fig. 5: Example of changes for the visitor case.

```
[public class ?visitedName                                1
   extends [...]@[(subtype*|sname ASTNode)]               2
     ?bodyVisited]@[(equals ?visited)]                    3

package be.ac.chaq.model.ast.java;                        4
?visitedImports                                           5
[?visited]@[match|set]                                    6

package be.ac.chaq.model.ast.java.visitor;                7
?visitorImports                                           8
public interface IASTVisitor {                            9
}                                                         10
=>                                                        11
[public void acceptVisitor(IASTVisitor visitor){          12
  visitor.(str "visit" ?visitedName)(this);               13
}                                                         14
]@[(add-element ?bodyVisited)]                            15

[import be.ac.chaq.model.ast.java.visitor.IASTVisitor;]   16
@[(add-element ?visitedImports)]                          17
[import be.ac.chaq.model.ast.java.?visitedName;]          18
@[(add-element ?visitorImports)]                          19

[public void (str "visit" ?visitedName)(?visitedName o);] 20
@[(add-element ?bodyVisitor)]                             21
```

Fig. 6: Specification for the visitor case.

wildcard on line 2 substitutes for the type extended by the class, which matching directive `subtype*|sname` requires to be a subtype of `ASTNode` or `ASTNode` itself. The template on lines 4–6 matches the compilation unit that declares this `?visited` class, together with its import declarations. Note that we could have also put lines 1–3 inside this template, similarly to the previous example. We chose not to in order to demonstrate how meta-variables can be used to compose templates.

The RHS of the specification uses the `add-element` rewriting directive to add the required method and import declarations. Some of these templates feature a Clojure expression that substitutes for a regular node. For instance, expression `(str "visit" ?visitedName)` evaluates to a string for the name of the method that is to be added to the visitor for each visited class. Users are responsible for ensuring that such expressions evaluate to a syntactically valid replacement value.

VI. RELATED WORK

Language-wise, the JUNGL [14] transformation language is closely related. It also advocates the use of functional programming (ML) for changing subjects identified through logic programming (Datalog), but does not feature code templates. It incorporates regular expressions over the paths through a control flow graph to express control flow characteristics. Our code templates support matching directive `match|regexp-path`

on statement lists to this end, using an EKEKO-based implementation [3] of path logic programming [15].

Purely functional or procedural transformation languages have been proposed as well. Famous examples include ASF [4], Stratego [5], and TXL [6]. Support for code templates in these systems is limited to syntactic characteristics. Purely logic-based transformation languages include JTL [9] and JTRANSFORMER [8]. JTL features a Java-like surface syntax for specifying syntactic and structural characteristics. However, none but the simplest specifications resemble actual Java code. JTRANSFORMER embeds actual code templates within logic formulas, but lacks the matching directives to support anything but syntactic characteristics. It also operates upon a logic fact base representing the program under transformation rather than the program itself. EKEKO's symbiosis with Eclipse [1] allows us to forego such an indirection.

Tool-wise, IXJ [10] for Java is the most closely related. Transformations are specified through an editor that visualizes the abstract grammar of code templates created from an initial code snippet. The editor features operations for introducing wildcards and meta-variables in a template, upon which a limited set of mostly type-related constraints can be imposed. As such, these templates lend themselves only to specifying syntactic characteristics of individual expressions or statements. Change actions are specified inline as after states for individual template elements, rather than through a rule with a LHS and RHS template. This limits its applicability to one-to-one rewrites of smaller code elements.

CHANGEFACTORY [11] for Smalltalk is also closely related tool-wise, while featuring transformation rules. These are specified through an editor starting from a recorded developer change. The subject of such a change is used as the seed for a code template, which can be refined by introducing meta-variables, wildcards and a limited set of syntactic matching directives. Support for changing the recorded changes themselves is limited.

The COCCINELLE [16] tool adopts the syntax of Unix patches with meta-variables for transformation specifications. A flow-based matching ensures that a single patch can be applied correctly to similar source code files. However, coarser-grained changes dispersed over several code elements such as required for a refactoring are not supported. More recently, several tools have been investigated for automatically inferring patch-like transformations from manually performed changes. We refer the reader to Kim et al. [17] for a recent survey.

VII. CONCLUSION

In this tool paper, we built EKEKO/X from the ground up starting from its applicative logic meta-programming foundation. Unique to EKEKO/X are its matching directives that provide fine-grained control over the way individual template elements are matched. We conjecture that these facilitate specifying transformations, starting from the code for an example subject and iteratively refining the resulting specification. We are currently exploring their benefits in the context of automatically generalizing recorded changes into a transformation.

ACKNOWLEDGMENTS

We thank Siltvani for her contributions to an early prototype in the context of her master's thesis [18]. This work has been supported, in part, by the Japan Society for the Promotion of Science, Kakenhi Kiban (S), No.25220003, by the Osaka University Program for Promoting International Joint Research, and by the *Cha-Q* project of the Flemish agency for Innovation by Science and Technology.

REFERENCES

[1] C. De Roover and R. Stevens, "Building development tools interactively using the Ekeko meta-programming library," in *Proceedings of the IEEE CSMR-WCRE 2014 Software Evolution Week, Tool Demo Track (CSMR-WCRE14)*, 2014.

[2] J. Fabry, C. De Roover, and V. Jonckers, "Aspectual source code analysis with GASR," in *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation (SCAM13)*, 2013.

[3] R. Stevens, C. De Roover, C. Noguera, and V. Jonckers, "A history querying tool and its application to detect multi-version refactorings," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR13)*, 2013.

[4] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier, "Compiling language definitions: the ASF+SDF compiler," *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 4, 2002.

[5] E. Visser, "Program transformation with Stratego/XT," in *Domain-Specific Program Generation*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 3016.

[6] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, 2006.

[7] Y. Padioleau, J. L. Lawall, and G. Muller, "SmPL: A domain-specific language for specifying collateral evolutions in linux device drivers," *Electronic Notes in Theoretical Computer Science*, vol. 166, 2006.

[8] G. Kniesel, J. Hannemann, and T. Rho, "A comparison of logic-based infrastructures for concern detection and extraction," in *Proceedings of the 3rd Workshop on Linking aspect technology and evolution (LATE07)*, 2007.

[9] T. Cohen, J. Y. Gil, and I. Maman, "Guarded program transformations using JTL," in *Proceedings of the 46th International Conference on Objects, Models, Components and Patterns (TOOLS08)*, 2008.

[10] M. Boshernitsan, S. L. Graham, and M. A. Hearst, "Aligning development tools with the way programmers think about code changes," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007.

[11] R. Robbes and M. Lanza, "Example-based program transformation," in *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems (MODELS08)*, 2008.

[12] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with eclipse," in *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ11)*, 2011.

[13] C. De Roover, C. Scholliers, V. Jonckers, J. Pérez, A. Murgia, and S. Demeyer, "The implementation of the Cha-Q meta-model: A comprehensive, change-centric software representation," in *Proceedings of the 8th International Workshop on Software Quality (SQM14)*, 2014.

[14] M. Verbaere, R. Ettinger, and O. de Moor, "JunGL: a scripting language for refactoring," in *Proceedings of the 28th International Conference on Software Engineering (ICSE06)*, 2006.

[15] S. Drape, O. de Moor, and G. Sittampalam, "Transforming the .NET intermediate language using path logic programming," in *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'02)*, 2002.

[16] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, and G. Muller, "A foundation for flow-based program matching: using temporal logic and model checking," in *Proceedings of the 36th Symposium on Principles of programming languages (POPL09)*, 2009.

[17] M. Kim and N. Meng, "Recommending program transformations," in *Recommendation Systems in Software Engineering*, M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds. Springer Berlin Heidelberg, 2014.

[18] Siltvani, "A workbench for template-driven program transformation," Master's thesis, Vrije Universiteit Brussel, July 2013.