Example-based Program Transformation

Romain Robbes and Michele Lanza

REVEAL @ Faculty of Informatics, University of Lugano, Switzerland
romain.robbes@lu.unisi.ch, michele.lanza@unisi.ch

Abstract. Software changes. During their life cycle, software systems experience a wide spectrum of changes, from minor modifications to major architectural shifts. Small-scale changes are usually performed with text editing and refactorings, while large-scale transformations require dedicated program transformation languages. For medium-scale transformations, both approaches have disadvantages. Manual modifications may require a myriad of similar yet not identical edits, leading to errors and omissions, while program transformation languages have a steep learning curve, and thus only pay off for large-scale transformations. We present a system supporting example-based program transformation. To define a transformation, a programmer performs an example change manually, feeds it into our system, and generalizes it to other application contexts. With time, a developer can build a palette of reusable medium-sized code transformations. We provide a detailed description of our approach and illustrate it with examples.

1 Introduction

Software developers have to continuously adapt their systems to new requirements, frameworks, operating systems, or even programming languages. These changes, often performed manually, are cumbersome and error-prone, especially if applied repetitively and/or transversely on a system. Several approaches have been introduced to automate changes. The program transformation spectrum can be roughly divided in:

Refactorings are well integrated in many IDEs and –generally being one right-click away– easy to apply. They are safe due to their behavior-preserving nature. They are widely used, and part of many a developers' toolbox. They are however limited in scope: Only the handful of most common refactorings are available in IDEs. Implementing a new refactoring involves a significant coding effort.

Linked Editing refers to the ability of some code editors to change a code fragment and have the editor broadcast the changes to similar regions of code. The code regions can be either documented or detected by the tool. The impact of such tools is however limited since they usually work at the text level, not the syntactic level, thus only applicable to code fragments with a high degree of similarity. Parametrized edits are seldom supported. Refactoring and linked editing form the small end of the spectrum.

Aspect-Oriented Programming (AOP) allows crosscutting concerns to be abstracted and separated from the code base into aspects. As part of the compilation process, the program is transformed to include the aspects which were extracted. However using AOP involves learning a new language on top of the regular one, and adding a tool to the toolchain. If a project is not using AOP from the start, developers might be reluctant to add it later on. Also, developers must learn to metaprogram, i.e., consider their programs as data structures, which is neither evident nor immediate.

Program Transformation Languages are the most flexible and powerful approach, but they involve learning a new language and reasoning about the program at another, more abstract, level. Tranformations tend to deal directly with the AST of the program, while AOP uses special purpose (and more limited) constructs such as advices and pointcuts. Such languages are seldom integrated in a development process, and applied to the entire program as a separate step of the build process, making the use worthwhile only for large-scale, program-wide, transformations. AOP and transformation languages form the large end of the spectrum.

In the middle of the transformation spectrum we find manually performed small to medium-scale transformations. These are quite common: Kerievsky [1] describes how to combine refactorings and manual editing to introduce design patterns in an existing system, covering more than twenty-five patterns. Other examples of such transformations are repetitive tasks such as adaptations of the system to changes related to the problem domain, especially in the presence of duplicated code. These are not well covered by existing approaches. In many cases developers opt to perform the changes manually, leading to further decay in the system.

We present an *example-based* program transformation approach: To specify how a repetitive task should be automated, a programmer *records* a sequence of changes as an example of it. The rationale behind our approach is that highly abstract activities such as defining program transformations have less overhead when one is working on concrete instances of a problem. Starting from the example working in its particular context, the developer generalizes it to make it applicable in other contexts. During this process, we allow the developer to directly interact with the structure of the transformation and with the entities affected by it. Finally, the programmer can explicitly name and store the newly defined program transformation, and reuse it whenever/wherever it is needed. The contributions of this paper are:

- An example-based program transformation approach, based on (1) the recording of a sequence of changes to provide the initial transformation structure to be worked on, (2) a direct interaction with this structure to refine and generalize it, and (3) the interaction with example entities to set the scope of the newly defined transformation.
- A proof-of-concept of our approach through two distinct examples that demonstrate its versatility: The definition of an “informal” logging aspect, and the definition of the “extract method with holes” refactoring.
- An implementation of the approach in the form of a “Change Factory”, enriching the Spyware platform that we built over the past few years.

Structure of the paper Section 2 presents our approach in a nutshell, while Section 3 introduces the running example we use in the remainder of the paper. Section 4 to Section 8 detail the individual steps of our approach. Section 9 discusses our approach, including its applicability to other models than program ASTs, while Section 10 covers related transformation approaches. Finally, we conclude in Section 11.

2 Transforming Programs by Example

Example-based program transformation relies on IDE monitoring to record exemplar changes, i.e., concrete instances of transformations. Monitoring is implemented as part of our SpyWare[2] platform for the Squeak Smalltalk IDE. SpyWare monitors the programmer's activity, and converts it to changes stored in a repository. These changes are reified first-class objects used by our transformation tool, the Change Factory. Using a graphical interface, a change is refined into a general-purpose program transformation. To help in testing the transformation, the Change Factory supports a quick edit/try cycle: At any time, the transformation can be compared against the original example.

Defining and using example-based program transformations is divided in 6 steps, described alongside an example (in italics), the definition of an informal logging aspect.

1. **Record changes.** Record a concrete example of a transformation, by performing it manually on example entities. Our monitoring plugin automatically converts developer activity into first-class changes. *The example change for a logging aspect consists in introducing a logging statement at the beginning of a method.*
2. **Generalize changes in a transformation.** This process is performed automatically given a concrete sequence of changes. Each entity reference in the change is converted to a variable. Based on how each entity is created, modified or removed in the change sequence, the system deduces a role for it. Some will be parameters to the transformation, while others will need to be computed from these parameters. *From the example change, the generalization process deduces that the change applies to a parameter, X, which is a method, and that the location where the statement is inserted must be specified by the user.*
3. **Edit variable parts of the transformation.** Based on the roles of the variables deduced from the previous step, the developer uses the Change Factory to edit the transformation and to specify how variables are computed. *The developer specifies where to insert the statement in X (at the beginning), and specifies that the string printed in the logging statement should contain the name of X.*
4. **Introduce higher-level structures.** The developer can introduce high-level constructs such as iterations or conditionals to build larger changes from elementary building blocks. This allows us to exploit another ability of our approach, which is to record any number of elementary changes for a tranformation. *The developer specifies that to apply the logging transformation to a class, the previous change must be applied to all the methods of the class. He can also define variants of the change, depending on the number of arguments in the calling method.*
5. **Test the transformation on examples.** At any time during steps 3 and 4, the developer can test the effects of the modified transformation by running it repeatedly on the example entities. *The developer compares the results of the initial change and the specified transformation on the same targets.*
6. **Apply the transformation to the system.** Once the transformation is defined, it is saved and can be immediately used from the code browser of the IDE. This allows the transformation to be applied to one entity at a time. The transformation can also be applied to a larger number of entities at once. *The logging transformation is stored, ready to be applied at any moment to any program. The transformation can also be undone.*

3 Running Example

We present a more complex example that we use in the following sections. The goal is to create a custom refactoring, called “Extract Method with Holes”, a more complex variant of “Extract Method”. According to Fowler a refactoring tool featuring “Extract Method” is probably complex enough to implement most refactorings[1]. Refactoring tools featuring “Extract Method” are able to infer which local variables need to be passed as arguments to the extracted block of code (those that are referenced both in the code block that is being extracted and outside of it). A frequent situation however is that a constant expression in the block of code would need to be passed as a parameter to the new method that is being created. Since this expression is referenced only inside the code block that is extracted, it is not converted to a parameter (Figure 1, top). Another related situation is when a method is called on an extracted variable. The call becomes part of the extracted code, while sometimes it should stay in the calling method (Figure 1, bottom).

initial code and selection	"Extract Method" behavior	desired behavior
	exampleMethod: argument	**exampleMethod:** argument
exampleMethod: argument	self addTo: argument.	self add: 42 to: argument.
argument + 42.	^ argument	^ argument
^ argument		
	addTo: argument	**add:** value **to:** argument
	argument + 42.	argument + value.
exampleMethod: arg1 **and**: arg2	**exampleMethod:** arg1 **and**: arg2	**exampleMethod:** arg1 **and:** arg2
arg1 + arg2 squared.	self add: arg1 to: arg2.	self add: arg1 to: arg2 squared.
^ arg1	^ argument	^ argument
	add: arg1 **to:** arg2	**add:** arg1 **to:** arg2
	arg1 + arg2 squared	arg1 + arg2.

Fig. 1. Default and desired behavior of “Extract Method’

In both cases, additional modifications are needed, using two possible alternatives:

1. **Extract Temporaries.** This is done by extracting the constant expression to a temporary variable, moving the temporary variable declaration out of code block, extracting the code block to the new method, and inlining the temporary again.
2. **Add Parameters.** This happens by extracting the code of the method, applying the “add parameter” refactoring to the newly created method, replacing the constant expression with the parameter in the body of the newly created method, editing the call site, and adding the constant expression in place of new parameter.

Both approaches require several steps and disrupt the flow of the programmer. In the following we show how, using our approach, we create the “Extract Method with Holes” refactoring: In addition to extracting a method, portions of constant code can be also extracted as parameters of the newly created method.

[1] See http://www.martinfowler.com/articles/refactoringRubicon.html

4 Recording the Example

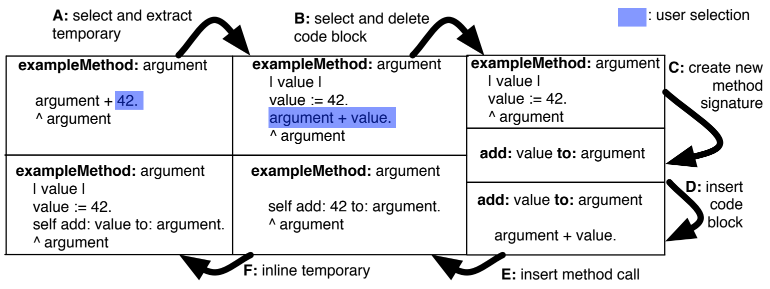
Recording changes is based on our previous work on *Change-Based Software Evolution* [3–5], whose key concept is to view *change as a first-class entity* by modeling software evolution as a sequence of changes that take a system from one state to the next. These changes are inferred from the developer activity recorded by the IDE's event notification system, whenever the developer modifies the system. In short, we do not view a software system's history as a sequence of versions, but as the sum of *change operations* which brought the system to its actual state. This is conceptually similar to operation-based versioning [6], but at a much finer-grained level.

Program Representation. We model a system as an evolving abstract syntax tree (AST) containing nodes representing packages, classes, methods, variables and statements. Nodes have *properties*, which vary depending on the node type, such as: for classes, name and superclass; for methods, name, return type and access modifier; for variables name, type and access modifier, etc. The name is a property of entities since identity is provided by a unique identifier (ID). Each AST entity has a *change history* containing all changes applied to it during the system's evolution.

Change Operations. Change operations represent the evolution of the monitored system: All changes a programmer performs are captured and reified. They represent transitions between states of an evolving system, i.e., adding/removing/changing classes/methods, refactorings, etc. We support *atomic* and *composite* change operations.

Atomic Changes. An atomic change is an operation on a program's AST. It is executable, and can be undone, since it contains all the necessary information to update the model by itself, and to compute its opposite atomic change. By iterating on the list of changes we can generate all the states a program went through during its evolution.

Composite Changes. Change operations can be abstracted into higher-level composite changes having an intuitive meaning for a developer. For example moving a class from package *A* to package *B* consists in removing it from *A* and adding it to *B*. These two atomic changes can be grouped in a single *move class* change.


A: select and extract temporary
B: select and delete code block
: user selection
C: create new method signature
D: insert code block
E: insert method call
F: inline temporary

```
exampleMethod: argument

   argument + 42.
   ^ argument
```

```
exampleMethod: argument
   | value |
   value := 42.
   argument + value.
   ^ argument
```

```
exampleMethod: argument
   | value |
   value := 42.
   ^ argument

add: value to: argument
```

```
add: value to: argument

   argument + value.
```

```
exampleMethod: argument

   self add: 42 to: argument.
   ^ argument
```

```
exampleMethod: argument
   | value |
   value := 42.
   self add: value to: argument.
   ^ argument
```

Fig. 2. Recorded changes for the running example

Recording Exemplified. For the "extract method with holes" refactoring, six composite changes (A - F) are recorded, as shown in Figure 2.

5 Generalizing the Example

Generalization is the process through which concrete changes are converted to transformations. In our system a transformation is a function that, given a set of parameters representing the entities to which the changes are applied, generates a new sequence of concrete changes similar to the example, but affecting arbitrary entities. To do this, we extended our change-based approach described previously with *generic changes*, which have the same structure as concrete changes, but their fields contain *variables*, instead of IDs of concrete entities. Whenever the variable is assigned an ID, all references to it are updated. To generalize an example, the developer looks into the change history, where the recent changes are stored, and selects the changes of interest.

Deducing the Role of Each Variable. Given the structure of a generic change, a role can be deduced for each variable, affecting how the programmer has to process it. For each variable *X*, these conditions are checked:

X is not created in the change. One of the changes affecting the variable is its creation. If X is not created in the change, its role is *parameter*. Otherwise, the variable does not belong to the parameters of the changes: A new, unique ID is generated for it each time the transformation is run.

X is an inserted/deleted statement. If X is inserted or deleted from the body of a method, a suitable location in the body of the method must be found when the transformation is run on a candidate method. X's role is thus *unlocated*.

X is inserted/deleted, but is not a subtree root. The location of X can be deduced from its parent's location. The programmer does not have to specify it, X is a *constant*.

Since changes can be composed, a given variable can play different roles in several parts of the change, e.g., it might be a parameter in some of the composite changes and a constant in another. At the transformation level, it will be a constant.

Generalizing Exemplified. Looking at the structure of the sequence of changes shown in Figure 2, the changes are affected the following way (the roles of variables are in italics):

Change A: The variable representing 42 is a *parameter*, which the user will set up via selection (see Section 6). The inserted statements (variable declaration and variable assignment), are *unlocated*.

Change B: The deleted block of code will also be a *parameter* of the transformation. Nodes under it are *constants*.

Change C: There are no parameters or variables needing a location, as every entity is created on the spot. They are *constants*.

Change D: The block of code is *unlocated* and needs a location (the very beginning of a method). The method which is a constant in C is a *parameter* in D.

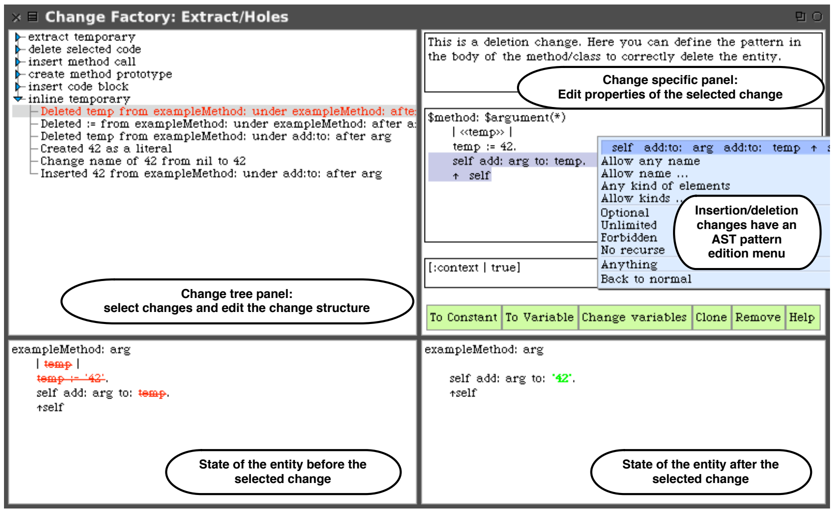
Change E: The *unlocated* method call needs a location (the previous position of the selected code block).

Change F: The deleted statements (variable declaration, assignment and reference) are *unlocated*, while the value in the assignment (also *unlocated*) needs to be relocated where the variable reference was.

In the overall transformation, the only *parameter* is the method the refactoring is applied to.

6 Editing the Example

Editing the example is the first step requiring manual work through a user interface: Figure 3 shows the Change Factory interface, made of 4 panels.


Change Factory: Extract/Holes
Change specific panel: Edit properties of the selected change
Insertion/deletion changes have an AST pattern edition menu
Change tree panel: select changes and edit the change structure
State of the entity before the selected change
State of the entity after the selected change
To Constant
To Variable
Change variables
Clone
Remove
Help

```
extract temporary
delete selected code
insert method call
create method prototype
insert code block
inline temporary
  Deleted temp from exampleMethod: under exampleMethod: afte
  Deleted := from exampleMethod: under exampleMethod: after a
  Deleted temp from exampleMethod: under add:to: after arg
  Created 42 as a literal
  Change name of 42 from nil to 42
  Inserted 42 from exampleMethod: under add:to: after arg
```

This is a deletion change. Here you can define the pattern in the body of the method/class to correctly delete the entity.

```
$method: $argument(*)
    | «temp» |
    temp := 42.
    self add: arg to: temp.
    ↑ self
```

```
self  add:to:  arg  add:to:  temp  ↑
Allow any name
Allow name ...
Any kind of elements
Allow kinds ...
Optional
Unlimited
Forbidden
No recurse
Anything
Back to normal
```

```
[:context | true]
```

```
exampleMethod: arg
    | ~~temp~~ |
    ~~temp := '42'~~.
    self add: arg to: ~~temp~~.
    ↑self
```

```
exampleMethod: arg

    self add: arg to: '42'.
    ↑self
```

Fig. 3. The Change Factory's main interface, shown editing a deletion change

Using this interface, one can (1) change properties of variables such as their name, (2) specify the location of unlocated variables via user selection or pattern matching, (3) specify conditions which may prevent or modify the action of the change, (4) modify the structure of the change by adding, removing or reordering changes (next section), (5) add control structures such as iterations and conditionals (next section). Table 1 sums up what properties are editable for each atomic change.

Atomic Change	Aim	Possible actions
Creation	Create a new entity of a given type.	Change the kind of entity created. Remove to convert a constant or a variable to a parameter. Create one to do the opposite.
Addition or Removal	Add/Remove a method, variable, class, package from the class, package, system.	Define a condition for the successful addition of the current class, package, method or variable.
Property Change	Change a property (name, superclass, etc.) of an entity.	Change the kind of property set. Change the value of the property set (constant or function). Add a success condition.
Insertion or Deletion	Inserts/Remove a statement-level entity in a method body.	Define the insertion/deletion AST pattern for an unlocated variable. Add a success condition. Specify if a selection should be used.

Table 1. The properties that can be edited for each change.

Changing Properties. By default, a generic property change keeps the property of the original concrete change, but the Change Factory allows it to be either a computation or a demand for user input (e.g., the user might want to pick a name or a superclass for a given class). To compute properties, the change factory provides the user with a *context* object that can be queried for information during the change's application. Using the context, one can access the values of the parameters and variables during the change execution, as well as the entire state of the program. One can assume all the changes before the current change in the tranformation are instantiated and executed. The context has a convenient API to access the most useful queries, such as: current class or method, current method name etc. Identifiers can be bound to values in the context. Those values can be retrieved later on, to transmit information between changes.

Specifying Conditions. The context can be used to define conditions altering the behavior of the change depending on where it is applied (e.g., if the current method does not override another). Conditions can be either pre-conditions (tested before the change takes place) or post-conditions (tested on the modified entities after the change takes place). If these are not met during instantiation, the change *fails*: All the concrete atomic changes generated by the change so far are undone.

Locating Entities. When methods are involved, "Unlocated" entities must be found in the methods to which the transformation will be applied. However, dealing with the intricacies of ASTs is one of the overly abstract activities a programmer faces when transforming programs. In addition, building a mental representation is hard since example run-time ASTs are not easily accessible. It is not clear which nodes to look for in the AST, and where they should be located. We adress these problems by using the ASTs of the concrete entities to which the recorded changes were applied to as an initial *AST pattern* to be incrementally refined by the programmer. An *AST pattern* is an AST enriched with information to relax or constrain its comparison to other ASTs. Furthermore, we minimize the need for the programmer to consider the AST pattern structure by providing a *direct manipulation interface*. These two features work in concert: For each insertion or deletion of an unlocated entity, the programmer is presented with the state of the example just before (for a deletion) or after (for an insertion) the change was applied (Figure 3, top right). The variable inserted or deleted is highlighted to ease focusing. Behind the printed text, the programmer is interacting with a serialized AST pattern of the state of the original example method. When the transformation is applied to another method, the AST of the method will be retrieved and matched against the AST pattern in an attempt to find either a correct place in the AST for an inserted entity, or the ID of an entity in the AST to delete. If the ASTs do not match, the change fails.

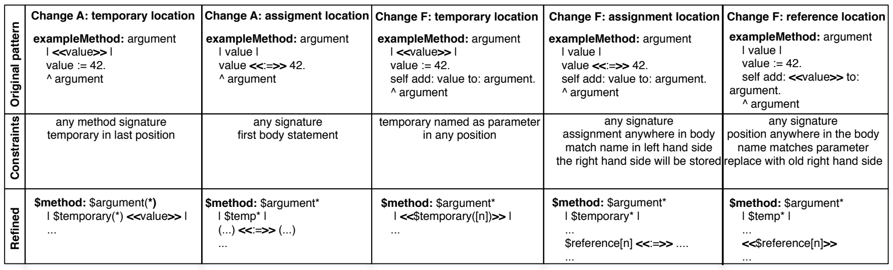
Refining patterns with constraints. If no modification is made to the AST pattern, it will only match the initial method in its initial state. We allow the programmer to relax the constraints in the AST pattern by simply selecting a node or a range of nodes in the text. The tool maintains a mapping from text positions to AST nodes, sparing the programmer to manually locate every node. A context menu gives the available constraints for the selected nodes. Upon selection, the constraints are applied to the nodes, and the text in the panel is re-rendered to reflect the updated constraints. Several constraints can be put on the same node (name, kind, multiplicity, optionality, recursion). The constraints are listed in Table 2. Some common sets of constraints are provided as shorthands, such as “allow any method signature” (a method having any name, an unlimited number of arguments and an unlimited number of temporaries), or “any position in the body” (inserts nodes in the pattern matching other nodes in the relevant position).

Constraint	Effect	Representation
None (default)	Same name and kind (entity type)	foo
Same kind	Same kind as original, any name is possible.	$temporary
Name matches	Same kind as original, name matches condition (e.g., the name must start with “set”).	$temporary([n])
Kinds	Matches any set of kinds (e.g., any body statement).	$k1|k2 or $*
Optional	May be present.	foo(?)
Forbidden	May not be present. (e.g., specify a counter example).	foo(!)
Unlimited	May be present several times.	foo(+) or foo(*)
No recursion	Ignore any children of the entity (e.g., the presence of an *if* is important, not its contents).	foo(. . .)
Anything	Any nodes (possibly none)	. . .
Manual	Specified by the programmer with Smalltalk code	foo([])

Table 2. Available constraints in AST patterns

Locating variables via selection. Some unlocated variables do not need an AST pattern to be found, but instead are selected by the user: When the change is instantiated, a window with the body of the method to which the insertion/deletion applies pops up, asking the user to select the relevant piece of code. One can also ask for the former position of a deleted selection (e.g., to substitute two pieces of code).

Editing Exemplified. Being selection-intensive, “Extract method with holes” does not need many property edits. The created variables in “Extract Temporary” (change A) need to be named, based on user input. In change C, the newly created method must be named –also via user input– based on the names of the arguments it takes. “Extract Method” must infer these arguments: It queries the context to get the set of arguments and temporaries which are referenced both inside and outside of the extracted block of code. The constant expressions and the extracted code block are user-selected. AST patterns need to be defined for the changes dealing with temporaries (A and F). Figure 4 shows how to refine the original AST to a satisfying AST pattern.



	Change A: temporary location	Change A: assigment location	Change F: temporary location	Change F: assignment location	Change F: reference location
Original pattern	**exampleMethod:** argument | **<<**value**>>** | value := 42. ^ argument	**exampleMethod:** argument | value | value **<<**:=**>>** 42. ^ argument	**exampleMethod:** argument | **<<**value**>>** | value := 42. self add: value to: argument. ^ argument	**exampleMethod:** argument | value | value **<<**:=**>>** 42. self add: value to: argument. ^ argument	**exampleMethod:** argument | value | value := 42. self add: **<<**value**>>** to: argument. ^ argument
Constraints	any method signature temporary in last position	any signature first body statement	temporary named as parameter in any position	any signature assignment anywhere in body match name in left hand side the right hand side will be stored	any signature position anywhere in the body name matches parameter replace with old right hand side
Refined	**$method:** $argument(*****) | $temporary(*) **<<**value**>>** | ...	**$method:** $argument* | $temp* | (...) **<<**:=**>>** (...) ...	**$method:** $argument* \| **<<**$temporary([n])**>>** | ...	**$method:** $argument* | $temporary* | ... $reference[n] **<<**:=**>>**	**$method:** $argument* | $temp* \| ... **<<**$reference[n]**>>** ...

Fig. 4. Initial patterns and resulting constraints

7 Composing Changes

The transformation we specified only allows one "hole" to be defined in the selection. How can we allow an arbitrary number of "holes" to be defined? We need to operate at a higher level than atomic changes: The Change Factory can edit the change structure itself. It allows one to (1) alter the change structure, (2) repeat one or more changes, (3) define conditional and optional changes, or (4) call other transformations.

Altering the Structure of the Transformation. The recorded changes and their generalized counterparts have a tree structure. The Change Factory allows us to alter this structure by adding, removing, or reordering the changes. Composite changes can also be merged (two composite changes become one with the atomic changes of both), or split (one composite becomes two). These alterations can affect the behavior of the change: Removing a creation changes the role of the created unlocated variable or constant to a parameter. Splitting a change allows to decompose a composite change in smaller steps. Some of those might be wrapped in conditional structures afterwards.

Iteration. Transformations without loops are limited. For instance, our example works only with the case of a single "hole". We allow a generic composite change (or more) to be applied to a set of entities via a generic *iteration change*. An *iteration set* of entities has to be named and defined via an initializer expression, using the context. The *iteration set* is then stored in the context, and can be reused by other iteration changes operating on the same entities. When instantiated, the changes inside the iteration are applied to each element of the set, in sequence, by binding parameters of the changes to all members of the set. For instance, a change taking a method as parameter can be converted to an iteration change taking as parameter a class and as initializer a context query returning all methods of the class. Alternatively, the *iteration set* members might not be parameters of the change per se, but used in computation of properties inside the change (e.g., the name of entities created in the change depends on the name of the entity being iterated on). When instantiated, the change binds the current entity to a name in the context. Changes needing it in the computation of properties can query the context by name. Another kind of iteration consists in attempting a change for an unspecified number of times, until it fails to apply (this last part being undone). Changes needing to match a variable occuring several times in a method can thus be simplified.

Alternative and Optional Changes. Sometimes, the correct change to apply might not be known until instantiation-time: Conditional changes are implemented by non-deterministically applying a sequence of changes and choosing the first one to apply correctly. Previous failing changes are undone. If all changes inside it fail, the conditional itself fails. Alternatively, a context query returning an integer i can be executed during instantiation of the conditional. The change at index i inside the conditional will be selected and executed. Optional changes are similar to the first variant of conditional changes. They attempt to apply the changes inside them, but they will never fail themselves.

Calling Other Transformations. Transformations can call one another. The calling transformation specifies the values of the parameters to give to the callee. This allows reuse of commonly used transformations as building blocks of bigger ones.

Composing Exemplified. When we recorded the changes to define "Extract Method, with Holes", our first selection of the changes contained an extra change, the creation of "exampleMethod:" itself. This change had to be removed to allow "exampleMethod:" to become a parameter to the transformation. Also during recording, changes C and D were performed as one single change, which had to be split in two.

To specify that an unlimited number of holes can be defined, extracted into temporaries and subsequently inlined, we used several iteration constructs: A set of temporary names T is initialized in change A using as many selections and input names as the user wants. At the end of the transformation, T is reused in change F to inline all the "temporary" temporaries. Change A initializes the set of temporaries by asking the user for selections until he stops. Also, in changes C and E, a set of arguments computed from the context –mentioned in the previous section– is used to build the signature of the extracted method and the call to it in the original method: Each argument is inserted once. For the sake of reuse, the extracted and inlined temporary transformations should be defined separately. Extract temporary takes as parameter a method, and asks the user to select an expression inside it, while Inline temporary takes as parameter a method and a temporary before inlining all references to the variable.

An alternative iteration is to define a fixed number of "holes", by calling several times the "extract temporary" change. At the end of the transformation, the calls to "inline temporary" can be wrapped in optional changes. It is improbable that many holes will be needed in extracted code, making this "brute force" solution viable.

8 Testing and Applying Transformations

Due to the exploratory nature of our tool, the boundary between testing and applying a change is fuzzy: An applied change is one which has not been undone.

How Changes Are Instantiated. Instantiating the change works as follows: (1) Evaluate pre-conditions, (2) Give actual values (IDs) to parameters, (3) Instantiate the transformation, (4) Apply the resulting changes to the code base, and (5) Evaluate post-conditions.

Each change generated by the transformation is stored in the change repository, where it is explicitely linked to the transformation that produced it. This allows each transformation application to be documented and assesses the impact of each transformation to the system. Should a transformation need to be changed, all its applications could be undone, and then redone with the new definition.

Applying a Transformation to the System. Once defined, transformations can be applied on a case-by-case basis or in a system-wide fashion.

Case-by-case. The transformation is named and stored in a repository, and made accessible within the IDE through menus. When browsing code, the user can decide to apply it to the entities he is currently viewing. This is the preferred way to use a transformation such as "extract method with holes".

System-wide. Other transformations can be applied on several entities using a Change Applicator tool, with which one can select a larger set of entities in the system and display the result.

9 Discussion

Example-based program transformations have the following characteristics:

Example-based. Using our change representation on top of the AST we can infer from a single example which entities are parameters to the change and which ones need to have their location defined for the change to be generalizable. Furthermore, the programmer uses direct manipulation of serialized ASTs to work with the concrete syntax of the system, instead of having to learn a dedicated syntax to match entities. Since those ASTs are also extracted from examples, one does not have to start over, only abstract away from the concrete example which was given.

Exploratory. In our change-based model of software evolution, changes are executable and can be undone. Exploring alternatives while designing the transformation is natural and backtracking is provided.

Compositional. We allow *sequences* of changes to be recorded and specified. This allows us to raise the level of abstraction from other pattern-matching based tools which usually operate on a single pattern. Thanks to change composition we can pass variables between changes and define iterations, in order to transform code on a larger scale, allowing us to define medium-scale transformations.

Documentation and evolution. The generic changes are stored in the change repository to be reused. The concrete changes they generate are documented as coming from the generic change: The rationale for the change is known. It is possible to undo these changes if the transformation is no longer suitable or if it needs to be modified. In that case, the updated transformation could then be reapplied to the same entities.

Applicability to other programming languages. Our prototype is implemented in Smalltalk, a language with a simple and consistent syntax. Applying it to a more complex language like Java which is typed and includes generic types, might pose some issues. We think it is feasible, since many transformation tools exist for Java: The existence of such tools might help us porting our approach. However, whether the approach is as usable in such a context is still undetermined.

Applicability to modelling languages. We believe our approach is also applicable for models. We transform programs, but consider them as ASTs and not simple text. As such, we transform a particular kind of models. Adapting our process for models in a language such as UML might actually be simpler, as there should be less need for matching AST patterns, which are more frequent when dealing with source code. In addition, change-based representations of models start to appear. Blanc et al. use a change-based representation of model to check their consistency [7]. Kögel [8] implemented an operation-recording versioning system for models, which could be used as a basis for our approach.

Quality of the examples. When coding, programmers often make errors and backtrack. These digressions are recorded in our changes as well and are unnecessary when generalizing the change. To adress this one can use our change-editing facilities to remove undesired changes from the sequence. Alternatively, the example can be "replayed", i.e., re-recorded to avoid the quirks introduced the first time around.

Behavior preservation. Unlike refactorings, behavior preservation is not guaranteed. Our tool requires programmer supervision to ensure the results are correct.

10 Related Work

Program transformation. Several domain-specific programming languages and systems exist to transform programs, addressing the problems of how to locate entities to be changed in the program (by parsing or pattern matching), and actually performing the transformation: The Design Maintenance System (DMS) by Baxter et al. [9] includes the Transformation Control Language (TCL). The DMS is designed to handle large transformations on large source code bases. One documented use of the DMS was the port of a large component-based C++ application to CORBA [10]. To gather the requirements of this transformation, the authors first converted one module of the application by hand. Stratego/XT is another well-known tool [11] for program transformation, which has seen applications in defining language extensions. In [12], Visser argues that manipulating ASTs of programs is too complex for many applications, and proposes a scheme to instead use the concrete syntax of the programming language. The result is a transformation language with both Stratego and the concrete syntax. In the same vein, De Roover et al. [13] introduced a concrete layer on top of a logic programming language, similar to Java source code, with variables to be matched prefixed by question marks. The rationale is also to simplify matching structures by hiding the AST where possible. Due to its extensive IDE integration, the language-based approach closest to ours is iXj by Boshernitsan et al. [14], an interactive IDE extension to transform Java programs supported by a visual programming language. ASTs are represented graphically. The transformation still has to be written with only the starting state specified (via selection).

Refactoring. Refactoring [15, 16] is a major feature of IDEs, which support the most common refactorings, first exemplified in the Smalltalk Refactoring Browser [17]. However the number of refactorings implemented in a refactoring engine is limited and covers only the most common cases. The same authors also implemented the Rewrite Tool [18] which uses pattern matching to implement arbitrary transformations. However the patterns defined in the transformations can only refer to one entity at a time and must be written from scratch in a dedicated language. Verbaere et al. have defined a dedicated scripting language [19], JunGL, to define new refactorings. JunGL is a hybrid functional/logic language whose logic layer eases writing queries on the AST. The "Extract Method" refactoring the authors describe still needs a few dozens of lines of code.

Model transformation. Transformation is a prominent concept in model-driven engineering. Several model transformation languages have been defined, such as MTL, Xion and Kermeta [20], or ATL [21]. However, they have a different target than our approach, since they transform abstract models of programs, and not the programs themselves.

While the approaches mentioned above are based on a program transformation language, a closer approach to ours is employed by Varró [22] to specify model transformations. His approach requires an example of a source model and a target model while we specifically consider changes between source-code entities. Another related approach has been documented by Wimmer et al. [23].

More lightweight approaches exist, such as regular expressions and keyboard macros in text editors. However, these techniques are text-based, losing the ability to easily query entities, and can be error prone.

Programming by example. The field of programming by example –or demonstration– (PBE) describes theories and provides tools which, from repetitive tasks performed by the user, infer a generalization of the behavior one wants to achieve. PBE aims for non-programmers to define simple automated tasks. PBE has been applied in a variety of areas [24]. In the same way, our system permits non-meta-programmers to define simple program transformations. However since the users differ, the approaches end up being quite different. Contrary to most PBE approaches, we do not wish to infer the action the user wants to do, but use the example to kick-start the generalization process. Since our users are programmers, they do not need a fully automatic process, but a sound basis to work with.

11 Conclusion

We presented an approach to specify program transformations which does not rely on a transformation language. Instead, its integration with the IDE and its underlying model of first-class changes allows programmers to define transformations by recording examples and subsequently generalize them. We defined a process supported by a tool, the Change Factory, to allow programmers to conveniently define transformations and where they should be applied in the system, based on the given examples. Several videos of the Change Factory in action are available at http://romain.robb.es/spyware/.

We illustrated the feasibility of our approach through two examples, refactoring definition and informal aspect definition. Our approach is also applicable in the domain of linked editing and code clone management, although space lacks to describe this here. We described our approach on the restricted set of program models (ASTs), but think it can be applied to more general models. Change-based modelling tools are starting to appear as research prototypes.

Our approach so far uses only one example to start the process. Having more examples could help to infer variables with more accuracy, and partially automate the definition of the AST patterns as well as where to apply changes in the system. This belongs to future work.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "REBASE" (SNF Project No. 115990).

References

1. Kerievsky, J.: Refactoring to Patterns. Pearson Higher Education (2004)
2. Robbes, R., Lanza, M.: Spyware: A change-aware development toolset. In: Proceedings of ICSE 2008 (30th International Conference in Software Engineering), ACM Press (2008) 847–850
3. Robbes, R., Lanza, M.: A change-based approach to software evolution. Electronic Notes in Theoretical Computer Science (ENTCS) **166** (2007) 93–109
4. Robbes, R., Lanza, M.: Characterizing and understanding development sessions. In: Proceedings of ICPC 2007 (15th International Conference on Program Comprehension), IEEE CS Press (2007) 155–164

5. Robbes, R., Lanza, M., Lungu, M.: An approach to software evolution based on semantic change. In: Proceedings of FASE 2007 (10th International Conference on Fundamental Approaches to Software Engineering). (2007) 27–41
6. Lippe, E., van Oosterom, N.: Operation-based merging. In: SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments, New York, NY, USA, ACM Press (1992) 78–87
7. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In Robby, ed.: ICSE, ACM (2008) 511–520
8. Kögel, M.: Towards software configuration management for unified models. In: CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models, New York, NY, USA, ACM (2008) 19–24
9. Baxter, I.D., Pidgeon, C., Mehlich, M.: Dms: Program transformations for practical scalable software evolution. In: ICSE, IEEE Computer Society (2004) 625–634
10. Akers, R.L., Baxter, I.D., Mehlich, M., Ellis, B.J., Luecke, K.R.: Reengineering c++ component models via automatic program transformation. In: WCRE, IEEE Computer Society (2005) 13–22
11. Visser, E.: Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer, C., et al., eds.: Domain-Specific Program Generation. Volume 3016 of Lecture Notes in Computer Science. Spinger-Verlag (2004) 216–238
12. Visser, E.: Meta-programming with concrete object syntax. In Batory, D., Consel, C., Taha, W., eds.: Generative Programming and Component Engineering (GPCE'02). Volume 2487 of Lecture Notes in Computer Science., Pittsburgh, PA, USA, Springer-Verlag (2002) 299–315
13. Roover, C.D., D'Hondt, T., Brichau, J., Noguera, C., Duchien, L.: Behavioral similarity matching using concrete source code templates in logic queries. In Ramalingam, G., Visser, E., eds.: PEPM, ACM (2007) 92–101
14. Boshernitsan, M., Graham, S.L., Hearst, M.A.: Aligning development tools with the way programmers think about code changes. In Rosson, M.B., Gilmore, D.J., eds.: CHI, ACM (2007) 567–576
15. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois (1992)
16. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
17. Roberts, D., Brant, J., Johnson, R.E., Opdyke, B.: An automated refactoring tool. In: Proceedings of ICAST '96, Chicago, IL. (1996)
18. Roberts, D., Brant, J.: Tools for making impossible changes - experiences with a tool for transforming large smalltalk programs. IEE Proceedings - Software **152**(2) (2004) 49–56
19. Verbaere, M., Ettinger, R., de Moor, O.: Jungl: a scripting language for refactoring. In Osterweil, L.J., Rombach, H.D., Soffa, M.L., eds.: ICSE, ACM (2006) 172–181
20. Muller, P.A., Fleurey, F., Vojtisek, D., Drey, Z., Pollet, D., Fondement, F., Studer, P., Jézéquel, J.M.: On executable meta-languages applied to model transformations. Model Transformations In Practice Workshop (2005)
21. Jouault, F., Kurtev, I.: Transforming models with atl. In Bruel, J.M., ed.: MoDELS Satellite Events. Volume 3844 of Lecture Notes in Computer Science., Springer (2005) 128–138
22. Varró, D.: Model transformation by example. In: Proc. Model Driven Engineering Languages and Systems (MODELS 2006). Volume 4199 of LNCS., Genova, Italy, Springer (2006) 410–424
23. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on (Jan. 2007) 285b–285b
24. Lieberman, H.: Your Wish Is My Command — Programming by Example. Morgan Kaufmann (2001)