Definition of language for specifying a transformation's subjects in an example-driven manner and its effects using first-class changes

Cha-Q project deliverable 3.1a

ANSYMO (Universiteit Antwerpen) — SOFT (Vrije Universiteit Brussel)

Responsible Coen De Roover (VUB)
Authors Coen De Roover (VUB), Christophe Scholliers (VUB)
Alessandro Murgia (UA), Javier Pérez (UA)

Contents

1 Introduction

This document reports on CHA-Q project Deliverable 3.1: the definition of language for specifying the subjects of a program transformation in an example-driven manner and its effects using first-class changes. The resulting program transformation language will facilitate repeating changes within and across different variants of a software system.

Developers often encounter this need. For instance, after having repaired one of several occurrences of a bug (i.e., semantic patching) or after having updated one of several clients of an API to a functionally equivalent one (i.e., API evolution and migration). Manually repeating such changes is not only laborious, but also error-prone.

1.1 Related Work

Specifying such changes as a generic source-to-source transformation enables automatic and safe reproduction, but doing so using one of the existing transformation specification languages requires too much expertise in practice. In general, program transformation languages adopt a set of rewrite rules for specifying a program transformation. The left-hand side (LHS) of such a rule identifies transformation subjects among the program's source code entities. The right-hand side (RHS) of the rule determines how the subjects have to be rewritten or changed by the transformations. Strategies determine the order in which these rules have to be applied —which can also be specified separately in the more advanced systems.

Functional approaches Visser presents a comprehensive survey of mainly functional transformation systems [1]. A great deal of them (e.g., Stratego [2] or ASF [3]) is based on term rewriting: each rewrite rule replaces a term that matches the one on its left-hand side by the term on its right-hand side.

Logic-based approaches Logic-based program transformation languages have also been proposed. In general, the left-hand sides of their rewrite rules consist of logic formulas over a fact base. The right-hand side specifies modifications to the fact base. The fact base of GENTL [4] and JTRANSFORMER [5] corresponds to a program's AST. This facilitates specifying the *structural characteristics* (i.e., instructions and their organisation) of a transformation's subjects through logic formulas.

TRANS [6] and path logic programming [7] operate upon the control flow graph (CFG) of a program instead. They support specifying the *control flow characteristics* (i.e., the order in which instructions may be executed at run-time) of a transformation's subjects: through temporal logic formulas and through regular expressions over the paths through the CFG respectively. The latter have also been incorporated in the ML-Datalog hybrid JUNGL [8] for specifying refactorings of C#-programs. Finally, DEEPWEAVER [9] supports expressing *data flow characteristics* (i.e., the values instructions may operate upon at run-time) explicitly through logic formulas over the results of a data flow analysis —albeit only an intra-procedural one.

Template-based approaches Some logic-based approaches feature a Java-like surface syntax for specifying structural characteristics. This partially addresses the shortcoming that specifying a subject's characteristics through logic formulas requires too much expertise. Among others, this is the case for JTL [10]. However, none but the simplest JTL specifications resemble actual Java code. JTRANSFORMER [5] is an evolution of the aforementioned GENTL that supports embedding actual code snippets within logic formulas. However, only structural characteristics can be specified in this manner.

Patch-based approaches SMPL [11] combines the traditional left-hand side and right-hand side of a rewrite rule into a single *semantic patch*. These extend the syntax of Unix patches with meta-variables and operators to abstract over irrelevant code. This way, a single patch can be applied correctly to similar source code files. They have been applied successfully to evolve client code to newer versions of an API [12] and to systematically correct common bugs [13]. However, semantic patches do not support coarser-grained changes that are commonly applied to object-oriented code. These would require transformation rules of which the left-hand side is not limited to control flow characteristics alone and of which the right-hand side supports a broader range of code updates.

The above approaches illustrate two important shortcomings of the state of the art in program transformation languages. First, they are tailored to specifying only one kind of characteristics of a transformation's subjects. Second, their use requires considerate expertise. Finally, existing specification languages result in transformations that cannot be applied in a change-aware manner. This is required when a program transformation has to be repeated on a system that is closely-related, but diverges from the one the transformation was intended for.

1.2 Requirements for the CHA-Q Transformation Formalism

To overcome the shortcomings of the state of the art, our specification language has to satisfy the following requirements:

R1: Adopt the CHA-Q meta-model for representing and applying changes. The CHA-Q meta-model represents changes to software artefacts as first-class objects that can be analyzed, repeated and reverted (cf. Deliverable 2.1.a). As such, a set of concrete `be.ac.chaq.change.Change` instances (i.e., a `be.ac.chaq.change.ChangeSet`) can already serve as a program transformation specification —albeit one that is restricted to predetermined change subjects.

A minimal requirement on the specification language is therefore to provide a means to refer to existing changes in a CHA-Q model (e.g., distilled (cf. Deliverable 2.2.a) or recorded from the history of a software system (cf. Deliverable 2.2.b), and a means to instantiate new changes for particular subjects. This will already enable developers to create a transformation from a concrete set of changes.

By convention, we will refer to the part of a CHA-Q transformation specification that determines what changes to perform as its right-hand side (RHS).

R2: Provide intuitive means for specifying and constraining change subjects. Repeating first-class changes on different (but closely-related) subjects requires a means to generalize a `be.ac.chaq.change.Change` to other model elements. A meta-variable `?subject`, for instance, could be used to substitute for the change's subject in a transformation specification. This renders the RHS of the specification a succinct representation of multiple changes —one per concrete binding `?subject`↦`modelelement` for the specification's `?subject` meta-variable.

Of course, these meta-variable bindings ought to be realized according to another developer-provided specification that determines the CHA-Q model elements that are subject to the transformation. We therefore require our specification language to provide a means to specify the structural (i.e., instructions and their organisation) and behavioural characteristics (i.e., the order in which instructions are executed at run-time and the values instructions operate on) of a transformation's subjects. By convention, we will refer to this part of a transformation specification as its left-hand side (LHS).

One proven option for this means is having developers formulate constraints over the subject meta-variable that are satisfied when the meta-variable is bound to the desired model element. For instance, the following constraints[1] could be used to restrict the meta-variable `?subject` to a model element that features as the single argument to a `be.ac.chaq.model.ast.java.MethodInvocation` of a particular `be.ac.chaq.model.ast.java.MethodDeclaration` named `setAge`:

```
invocation_has_argument(?inv, ?subject),
invocation_calls_method(?inv, ?method),
method_has_name(?method, "setAge")
```

Realizing the meta-variable's binding then amounts to solving a constraint satisfaction problem that ranges over all model elements —clearly something to be left to the transformation system rather than the developer.

A more intuitive option for this means is having developers use the subject meta-variable in a code template (i.e., a code snippet in which the meta-variable functions as a cutout). For instance, the code template `?receiver.setAge(?subject)` could serve the same purpose as the aforementioned constraints. Realizing the meta-variable's binding then amounts to matching a template against model elements — again something to be left to the transformation system.

R3: Enable change-aware transformation applications Assuming that we have an intuitive specification of program transformations, the next challenge is applying such program transformations in a change-aware manner. This is required when a program transformation has to be repeated on a system that is closely-related, but diverges from the one the transformation was intended for. The CHA-Q project is therefore to investigate

[1] We are using a Prolog-like notation for this hypothetical constraint language.

techniques that enable applying a transformation such that the changes between the systems are taken into account.

Deliverable 3.2.a will report on a technique for determining the impact of changes on existing program transformations. Deliverable 3.2.b will report on tool support for changing transformation specifications accordingly. While these deliverables are only due in Q6 and Q12 respectively, the CHA-Q transformation language should not preclude tool-supported changes to transformation specifications and where possible already facilitate them.

1.3 Design of the CHA-Q Transformation Formalism

In this deliverable, we specify the CHAQEKO/X transformation specification language of the CHA-Q project. This language will satisfy the above requirements as follows:

R1: Functional expressions involving CHA-Q changes as the RHS To satisfy requirement **R1** (*Adopt the* CHA-Q *meta-model for representing and applying changes*) we advocate founding the CHA-Q specification formalism on a functional language with seamless Java interoperability. Our motivation is two-fold. On the one hand, functional programing languages have proven themselves for transformation tasks (cf. Section 1.1). Here, higher-order functions enable composing transformation specifications in a natural manner. On the other hand, our CHA-Q meta-model already defines an object-oriented representation of software systems (cf. Deliverable 2.1.a) and has been implemented in Java (cf. Deliverable 2.1.b).

Clojure represents the most promising functional programming language to explore as the foundation for the CHA-Q specification formalism. First of all, it features the required interoperability with Java; methods can be invoked on Java objects from within functional expressions. To illustrate, the following two-argument Clojure function changes a CHA-Q AST node `subject` that represents a Java `int` literal (e.g., `5`) into a CHA-Q AST node that represents the corresponding `Integer`-creating expression (e.g., `new Integer(5)`):

```
1  (defn change-int-to-integer [snapshot subject]
2    (let [creationChange
3          (EntityCreationChange. snapshot ClassInstanceCreation)
4          createdEntityPlaceHolder
5          (.getCreatedEntity creationChange)
6          argumentListChange
7          (AddElementToListPropertyChange.
8            snapshot
9            createdEntityPlaceHolder
10           "arguments"
11           subject)
12         replacementChange
13         (EntityReplacementChange. snapshot subject createdEntityPlaceHolder)]
14     (.perform creationChange)
15     (.perform argumentListChange)
16     (.perform replacementChange)))
```

Here, lines 2–3 instantiate a new change object `creationChange` by invoking the constructor of class `be.ac.chaq.change.EntityCreationChange` with the required arguments.[2] The first argument corresponds to the snapshot in which the change is to be performed. The second argument corresponds to the particular `ASTNode` subclass of which the change object is to create an instance. Lines 4–5 retrieve a placeholder `createdEntityPlaceHolder` for the entity that will be created once the change is performed. To this end, they invoke method `getCreatedEntity()` on the `creationChange` object.[3] Lines 6–11 instantiate another change object `argumentListChange` that is to add the `int`-representing `subject` to the list of arguments to `createdEntityPlaceHolder` —thus creating an expression `new Integer(subject)`. Lines 12–13 instantiate a change object `replacementChange` that will replace `subject` by `createdEntityPlaceHolder` in its own parent AST node. Finally, lines 13-16 perform all created changes. This example illustrates instantiating and performing first-class CHA-Q changes from within Clojure functions. Our specification language will provide a library of similarly-defined functions that perform higher-level changes in terms of the corresponding low-level ones supported by the CHA-Q meta-model.

Next to its support for seamless Java interoperability, another benefit of Clojure is its support for managing the complexity that comes with concurrent manipulations of CHA-Q models. Examples include actor-like agents as well as software transactional memory. The need for concurrent manipulations of a CHA-Q model might arrive in the future when multiple developers start analyzing and applying model changes at the same time, but also when individual model manipulations need to be scaled up to industrially-sized projects.

Finally, like most Lisp derivatives, Clojure features a read-eval-print loop on which developers can evaluate and inspect the results of expressions in an interactive manner. This naturally enables an exploratory specification style in which transformations are refined and generalized incrementally.

R2: Constraints involving code templates as the LHS To satisfy requirement **R2** (*Provide intuitive means for specifying and constraining change subjects*), we propose to found the left-hand side of the CHA-Q specification formalism on a seamless embedding of constraint logic programming within functional Clojure expressions. Such an embedding already forms the foundation for our EKEKO [14] and EKEKO/X [15] libraries that enable querying and manipulating an Eclipse workspace respectively. They have proven themselves for answering program queries (e.g., "*does this bug pattern occur in my code?*"), for analyzing project corpora (e.g., "*how often does this API usage pattern occur in this corpus?*"), and for transforming programs (e.g., "*change occurrences of this pattern as follows*"). All of these applications can be formulated as a constraint satisfaction problem of which the solutions are processed by a functional expression.

The required embedding of constraint logic programming within functional programming can be provided by the CORE.LOGIC[4] port to Clojure of MINIKANREN [16]. The following Clojure expression illustrates this embedding. Note that `(for [<var> <exp_coll>] <body_exp>)` is the Clojure list comprehension built-in. It evaluates

[2] The `(<NameOfClass>. <arg_1> ... <arg_n>)` syntax instantiates a Java class from within Clojure.

[3] The `(.<nameOfMethod>. <receiver> <arg_1> ... <arg_n>)` syntax in Clojure is equivalent to the `<receiver>.<nameOfMethod>(<arg_1>, ..., <arg_n>)` syntax in Java.

[4] https://github.com/clojure/core.logic

<body_exp> for every element <var> in the collection <exp_coll>:

```
1 (for [subject
2         (run [?subject]
3           (fresh [?inv]
4             (methodinvocation-methoddeclaration ?inv ?method)
5             (has :name ?method  "setAge")
6             (child :arguments ?inv ?inv ?subject)))]
7   (change-int-to-integer ?subject))
```

Lines 2–6 (i.e., <exp_coll> of the list comprehension) roughly correspond to the logic constraints from the previous section. They restrict the bindings for meta-variable ?subject to the argument of a setAge() invocation. Section 2.1.1 discusses how the transformation system finds solution for these constraints. For now, it suffices to understand that lines 2–6 evaluate to a collection of bindings for ?subject. Line 1 (i.e., <var> of the list comprehension) and line 7 (i.e., <exp_coll> of the list comprehension) apply the change-int-to-integer function defined above to each ?subject↦subject binding found in this manner.

While constraint logic programming can be effective for restricting the subjects of a transformation, it is far from intuitive. Our language is therefore required to support code templates as a means to specify *both* structural (i.e., instructions and their organisation) and behavioural (i.e., the order in which instructions are executed at run-time and the values instructions operate on) characteristics of a transformations subjects. As constraint logic programming already provides the foundation for the left-hand side of the CHA-Q transformation language, we propose to embed code templates within specifications as an additional kind of logic constraint. For instance, through a logic constraint (match <ast> <template>) that is satisfied for every <ast> from the program under transformation that matches the given code <template>. Here, <template> is concrete syntax (e.g., a snippet of Java or Smalltalk code) in which meta-variables substitute for unknowns. We have used similar embeddings in the SOUL [17] and EKEKO/X [15] specification languages. Using a code template, our transformation can be re-specified as follows:

```
(for [subject
        (run [?subject]
          (fresh [?inv ?receiver]
            (match ?inv "?receiver.setAge(?subject)")))]
  (change-int-to-integer ?subject)
```

Whether an AST node matches a code template depends on a particular matching strategy. Sensible defaults have been shown to give rise to an intuitive example-driven matching that is geared towards detecting implementation variants of a code template [18]. To prevent unwarranted transformation applications, however, the CHA-Q specification language requires a means for developers to exert more control over the way templates are matched.

Based on our experiences with EKEKO/X [15], we propose the syntax [<template-component>]@[<directive-name$_1$> ...<directive-name$_n$>] for overriding the default matching strategy for individual template components. Here, each <directive-name$_i$> will be imposed implicitly on future matches for

<template-component>. Matching directives allow us to revisit the transformation specification as follows:

```
(for [subject
      (run [?subject]
        (fresh [?inv ?receiver]
          (match ?inv "[?receiver]@[relax-receiver].setAge(?subject)")))]
  (change-int-to-integer ?subject)
```

Here, we have added an relax-receiver constraint that relaxes the default matching strategy for method invocations. Indeed, the dot in the concrete syntax "?receiver.setAge(?subject)" inherently disallows matches without a receiver. The newly-added constraint ensures that an invocation setAge(5) with an implicit this receiver will match our code template. Matching directives can also take additional arguments using the notation (<directive-name> <arg$_1$> ...<arg$_n$>). Note that such directives receive the match for the component they are imposed on as an implicit <arg$_0$> argument.

Implementation-wise, a (match <ast> <template>) constraint can be expanded at compile-time or at run-time into regular constraints on its first argument and on the meta-variables in its code template. The expanded constraints determine whether <ast> matches <template> according to the latter's matching directives.

Finally, similar ideas can be applied to the right-hand side of a transformation specification. For instance, by generating code for a code template on the RHS using meta-variable bindings established by the LHS. Instead of matching directives, rewriting directives [<template-component>]@[<directive-name$_1$> ...<directive-name$_n$>] can be associated with the components of such a template. Instead of being expanded into logic constraints, such a rewriting directive is expanded into functional expressions that instantiate and apply the appropriate CHA-Q change objects. Note that like matching directives, they can take additional arguments using the notation (<directive-name> <arg$_1$> ...<arg$_n$>). However, they receive newly generated code as an implicit <arg$_0$> argument rather than a template match.

A Clojure macro (chaqeko/x <LHS> <RHS>) renders our final specification succinct:

```
(chaqeko/x
   [?receiver]@[relax-receiver].setAge(?subject)
   =>
   [new Integer(?subject)]@[(replace ?subject)])
```

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

R3: Operators for changing code templates To comply with requirement **R3** (*Enable change-aware transformation applications*), our language should define a representation of specifications that is easily analyzed and manipulated by external tools —in particular the tool support envisioned for Deliverable 3.2.a (i.e., a tool for computing the impact that code changes have on existing transformations) and for Deliverable 3.2.b (i.e., a tool for changing the transformation accordingly).

The latter tool support will be based on a suite of operators. When applied to a code template, the envisioned operators change the template such that it matches different code entities. For instance, an operator `(rename-method <template> <name> <newname>)` could change all occurrences of a method that has been renamed. Similar operators could be used to introduce additional meta-variables in a specification. This enables abstracting over the differences between the system a specification was intended for and the system it is being applied to. Finally, an operator could also be used to generalize the code template such that it matches more code entities, or to refine the code template such that it matches less code entities. Given a suitable template representation, this merely requires changing the matching strategies for the appropriate template component. Note that the envisioned operators give rise to transformation language of their own.

2

ChaQeko/X: The Cha-Q Transformation Formalism

In the previous chapter, we specified several requirements for the Cha-Q transformation language. We also motivated some initial choices in the design of ChaQeko/X that ensure these requirements are fulfilled. As in most approaches to program transformation, our 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 the identified subjects should be changed.

The design choices we discussed culminate in a (chaqeko/x <LHS> <RHS>) macro. To recapitulate, the following ChaQeko/X transformation changes the int argument of setAge invocations into a corresponding Integer argument:

```
(chaqeko/x
   [?receiver]@[relax-receiver].setAge(?subject)
   =>
   [new Integer(?subject)]@[(replace ?subject)])
```

Implementation-wise, the chaqekeko/x macro expands <RHS> into a functional expression and <LHS> into a constraint logic problem. The former applies first-class Cha-Q changes to all solutions for the latter.

Note that we deliberately introduced our specification language from the ground up. This is because it is important for the language to support the intermediate specifications as well. Their constructs offer explicit control over transformation application strategies — something that might be required for more involved use cases. In this chapter, we detail those functional and constraint logic programming constructs.

2.1 Applicative Declarative Foundations

ChaQeko/X owes its name to the Ekeko [14] and Ekeko/X [15] libraries for Clojure that enable querying and manipulating an Eclipse workspace respectively. ChaQeko/X brings this style of meta-programming to instances of our Cha-Q meta-model (i.e., an object-oriented representation of all of a system's artefacts and their changes [19]).

Our experiences with these libraries have influenced several choices in ChaQeko/X's design. Most notably, we forego a transcription to logic facts for reifying model elements as logic data. Such a transcription hampers perusing query solutions within tools. Instead, we leave the reified version of a Cha-Q AST node as the AST node itself (i.e., an instance of be.ac.chaq.model.ast.java.ASTNode). Embedding our logic language within Clojure, which offers excellent Java interoperability, enables this identity-based reification.

2.1.1 An Embedding of Constraint Logic Programming in Clojure

EKEKO, EKEKO/X and CHAQEKO/X share the CORE.LOGIC[1] port to Clojure of MINIKANREN [16] as their logic foundation. Queries can be launched from the Clojure read-eval-print loop using the chaqeko special form. It takes a vector of logic variables, each denoted by a starting question mark, followed by a sequence of logic goals:

```
(chaqeko [?x ?y]
  (contains [1 2] ?x)
  (contains [3 4] ?y))
```

Binary predicate contains/2, used in both goals, holds if its first argument is a collection that contains the second argument. Solutions to a query consist of the different bindings for its variables such that all logic goals succeed. Internally, the logic engine performs an exploration of all possible results, using backtracking to yield the different bindings for logic variables. The four solutions to the above query consist of bindings [?x ?y] such that ?x is an element of vector [1 2] and ?y is an element of vector [3 4]: ([1 3] [1 4] [2 3] [2 4]).

Logic variables have to be introduced explicitly into each lexical scope. Above, the chaqeko special form introduced two variables into the scope of its logic conditions. Additional variables can be introduced through the fresh special form:

```
1 (chaqeko [?x]
2   (differs ?x 4)
3   (fresh [?y]
4     (equals ?y ?x)
5     (contains [3 4] ?y)))
```

The above query has but one solution: ([3]). Indeed, 3 is the only binding for ?x such that all goals succeed. The differs/2 goal on line 2 imposes a disequality constraint such that any binding for ?x has to differ from 4. The equals/2 goal on line 4 requires ?x and the newly introduced ?y to unify.

2.1.2 Reification of CHA-Q Models

CHAQEKO/X provides a library of predicates that can be used to query the information that is maintained by the CHA-Q models. We limit our discussion to those predicates that reify structural relations about and between CHA-Q AST nodes.[2]

Binary predicate (ast ?kind ?node), for instance, reifies the relation of all AST nodes of a particular type. Here, ?kind is a Clojure keyword denoting the capitalized, unqualified name of ?node's class. Solutions to the query (ekeko [?inv] (ast :MethodInvocation ?inv)) therefore comprise all method invocations in the source code.

Ternary predicate (has ?propertyname ?node ?value) reifies the relation between an AST node and the value of one of its properties. Here,

[1] https://github.com/clojure/core.logic

[2] Similar predicates reify the relations involving CHA-Q changes, commit messages, and issues.

`?propertyname` is a Clojure keyword denoting the decapitalized name of the property's `be.ac.chaq.model.entity.EntityPropertyDescriptor` (e.g., `:modifiers`). In general, `?value` is either another `ASTNode` or a wrapper for primitive values and collections. This wrapper ensures the relationality of the predicate. The following query will therefore retrieve nodes that have `null` as the value for their `:expression` property (e.g., invocations with an implicit `this` receiver):

```
(ekeko [?node]
  (fresh [?exp]
    (nullvalue ?exp)
    (has :expression ?node ?exp))
```

Finally, the `child/3` predicate reifies the relation between an AST node and one of its immediate AST node children. Solutions to the following query therefore consist of pairs of a method invocation and one of its arguments:

```
(ekeko [?inv ?arg]
  (ast :MethodInvocation ?inv)
  (child :arguments ?inv ?arg))
```

Conceptually, the implementation of these lower-level predicates uses the aforementioned `contains/2` over an AST node relation maintained by each CHA-Q model. Our implementation of the higher-level predicates illustrates that CORE.LOGIC (cf. Section 2.1.1) embeds logic programming within a functional language. Binary predicate `child+/2`, for instance, can be implemented as a regular Clojure function that returns a logic goal:

```
(defn child+ [?node ?child]
  (fresh [?keyw ?ch]
    (child ?keyw ?node ?ch)
    (conde [(equals ?ch ?child)]
           [(child+ ?ch ?child)])))
```

Here, special form `conde` returns a goal that is the disjunction of one or more goals. Predicate `child+/2` therefore reifies the transitive closure of the `child/2` relation.[3]

Next to facilitating the use of query results in tools and preventing queries from returning stale results that no longer reflect the current state of the workspace, our identity-based reification of AST nodes (cf. Section **??**) also brings along some practical implementation advantages. Many predicates call out to Java whenever this is more convenient than a purely declarative implementation:

```
(defn ast-parent [?node ?parent]
  (fresh [?kind]
    (ast ?node ?ast)
    (differs null ?parent)
    (equals ?parent (.getParent ?node))))
```

Here, the last line ensures that `?parent` unifies with the result of invoking method `ASTNode.getParent()` on the binding for `?parent`. The before-last line ensures that the predicate fails for `CompilationUnit` instances which function as the root of the AST.

[3]Note that an idiomatic Prolog definition would consist of two rules that define the same predicate: one for the base case and one for the recursive case, thus creating an implicit choice point.

2.2 Code Templates for Specifying Transformation Subjects

Section 1.3 introduced code templates as a more intuitive means for specifying the subjects of a transformation. Within CHAQEKO/X, these take the form of a logic constraint (match <ast> <template>) that is satisfied for every <ast> from the program that matches the given code <template>. Solutions to the following query therefore consist of all return null; statements from the program under transformation:

```
(chaqeko [?s]
  (match ?s "return null;"))
```

Meta-variables can be used within a code template to substitute for unknowns. Solutions to the following query are therefore pairs of a return statement and its expression component:

```
(chaqeko [?s ?e]
  (match ?s "return ?e;"))
```

Of course, bindings are also realized for the meta-variables within a code template (e.g., [?s↦return null; ?e↦null] is included among the solutions to the query).

Finally, matching directives <directive-name> or (<directive-name> <arg$_1$> ...<arg$_n$>) can be associated with individual components of a template using a [<template-component>]@[<matching-directive$_1$> ...<matching-directive$_n$>] syntax. The relax-expression directive, for instance, ensures that the following template also matches return; statements —leaving ?e unbound:

```
(chaqeko [?s ?e]
  (match ?s "[return ?e;]@[relax-expression]"))
```

Without this directive, the concrete syntax of the code template would disallow such matches. The same effect could also be achieved using a disjunction of the two concrete syntax cases:

```
(chaqeko [?s ?e]
  (conde [(match ?s "return ?e;")]
         [(match ?s "return;")]))
```

Note that all directives receive a placeholder for future matches of the template component as an implicit <arg$_0$> argument. Implementation-wise, this placeholder is an internal meta-variable that substitutes for the match —hinting at the fact that matching directives are names of logic constraints.

The following is a preliminary list of matching directives that CHAQEKO/X ought to support. It is based on our experiences with the embeddings of code templates in SOUL [17] and EKEKO/X [15]. We will revisit and possibly expand this list in our report on the prototype implementation of CHAQEKO/X (Deliverable D3.2.a).

Matching directives for equality and disequality

- Matching directive equals/1 corresponds to the logic constraint equals/2 discussed in Section 2.1.1. Used within a logic query, it imposes a unification constraint between its arguments. Used within a code template, it imposes this constraint between its explicit and implicit argument (i.e., the match for the component it is associated with). Solutions to the following query therefore consist of bindings [`?s`↦`return ?a + ?b;` `?e`↦`?a + ?b;`]:

```
(chaqeko [?s ?e]
  (fresh [?a ?b]
    (match ?s "return [?a + ?b]@[(equals ?e)];")))
```

The `equals/2` directive enables retrieving the match for a template component (e.g., ?e) that is already constrained by concrete syntax (e.g., `?a` + `?b`).

- Matching directive `differs/1` corresponds to the logic constraint `differs/2` discussed in Section 2.1.1. It imposes a disequality constraint between its arguments, meaning that both should never unify. It is often convenient for ruling out duplicates among a transformation's subjects.

Matching directives involving containment

- Matching directive `parent/1` corresponds to the `ast-parent/2` logic constraint from Section 2.1.1. It ensures that the parent node of its implicit match argument is its explicit argument. The following query uses this directive to retrieve the statement in which an assignment expression resides:

```
(chaqeko [?s ?e]
  (fresh [?lhs = ?rhs]
    (match ?e "[?lhs = ?rhs]@[(parent ?s)]")))
```

  A related zero-argument `parent/0` directive is the default (and therefore hidden) matching directive for all template components: given an AST node N_p of a program, a template component N_t, and P_t as the parent of N_t; N_p matches N_t if there exists P_p where P_p matches P_t, and N_p is a child of P_p.

- Matching directive `parent+/1` is the transitive closure variant of the above `parent/1`. As such, it ensures that the component it is associated with has its explicitly specified argument as an ancestor.

  More interestingly, the related zero-argument `parent+/0` directive can be used to specify that the match for a template component can reside at an arbitrary depth within the match for its parent template component.

- Zero-argument matching directives `contains/0`, `contains-lexical/0` and `contains-flow/0` are in a sense dual to the aforementioned ones as they are meant to be associated with "parent" template components such as method declaration and class declaration templates.

  The `contains/0` directive imposes a containment constraint between the match for a template component and the matches for its children. Solutions to the following query therefore consist of any pair of statements from a `method()`:

```
(chaqeko [?s1 ?s2}]
  (fresh [?m]
    (match ?m
       "[public void method() {
          ?s1
          ?s2
      }]@[contains]")))
```

Note that the imposed containment constraint implies that ?m may contain additional statements for which no counterpart exists in the template. Furthermore, a pair of bindings [?s1↦s1 ?s2↦s2] does not entail that s1 precedes s2 in method(). Where this is desirable, matching directive contains-lexical/0 can be used instead. Similarly, contains-flow/0 only allows bindings [?s1↦s1 ?s2↦s2] such that s1 may be executed before s2 (i.e., there should be path in the control flow graph of method() between them).

Semantic matching directives With the exception of contains-flow/0, all of the above directives concern information maintained by the be.ac.chaq.model.ast.java.ASTNode hierarchy of our CHA-Q meta-model. The following matching directives concern information maintained by the be.ac.chaq.model.famix.SourcedEntity hierarchy (i.e., those derived from the original FAMIX model) instead:

- Matching directive qualified-name/1 can be associated with named entities in template components such as type references, type declarations, variable references, variable declarations, and method declarations. It takes as its sole explicitly specified argument a string that corresponds to the fully qualified name of the match for the template component it is associated with. Solutions to the following query would therefore consist of both fully and unqualified references to the type java.lang.Integer within a local variable declaration.

```
(chaqeko [?typeref]
  (fresh [?var ?localvardeclaration]
   (match ?localvardeclaration
     "[?typeref]@[(qualified-name 'java.lang.Integer')] ?var;")))
```

  The related relax-name/0 directive, in contrast, allows unqualified references in a match to a type that is fully qualified in the template:

```
(chaqeko [?localvardeclaration]
  (fresh [?var]
    (match ?localvardeclaration "[java.lang.Integer]@[relax-name] ?var;")))
```

  Likewise, the relax-type/1 directive allows matches to use a subtype of the type that is specified in the template —expect for method return types. To respect contravariance, relax-type/1 allows super types rather than subtypes for method return types.

- Matching directives may-alias/1 and must-alias/1 associate aliasing constraints between two expressions. These constraints are satisfied when the expressions may or must evaluate to the same value at run-time according to a dataflow analysis.

> As such an analysis is often expensive to compute, we also offer directives `references/1` and `declares/1` that impose constraints between a variable (or type) declaration and a reference to the variable (or type) they declare. The related matching directives `invokes/1` and `invokedby/1` impose constraints between a method invocation (or its name) and one of the method declarations (or their names) they might invoke at run-time.

The following query illustrates how the all of these matching directives can be combined. Its solutions consist of getter methods for `java.util.LinkedList` subtypes:

```
(chaqeko [?getter]
  (fresh [?field ?classname ?typedeclaration ?type ?varref]
    (match ?typedeclaration
      "[public class ?classname {
         private [java.util.LinkedList]@[relax-name relax-type (equals ?t)]  ?field;

         public  [?type]@[(relax-type ?t)] ?getter() {
            return ?varref@[(references ?field)];
         }
       }]@[contains]")))
```

At this point, one might argue that specifying our code templates is far from intuitive. We therefore advocate finding sensible default matching directives for each kind of template component. Such defaults have been shown to result in an example-driven specification style that enables detecting implementation variants of an exemplified code snippet [18]. However, we conjecture that the control offered by ChaQeko/X over template matching will be required for the more involved program transformation problems. After all, a default matching strategy that results in unwanted matches for a transformation's subjects will result in erroneous changes.

2.3 Code Templates for Specifying Transformation Effects

Having detailed the left-hand side of ChaQeko/X transformation specifications, we now turn our attention to the right-hand side; i.e., the side that specifies how each identified transformation subject is to be rewritten.

As explained in Section 1.3, code templates within constraint logic problems lend themselves for specifying the subjects of a transformation. Embedding constraint logic programming within functional programming naturally enables rewriting subjects identified in this manner using functional expressions that invoke our Cha-Q change API. Clojure's interoperability with Java enables us to provide functions to remove (`remove`), insert (`add`) and replace (`replace`) AST nodes. A function `change-property`) should enable changing the values of their properties. Here, properties can be denoted using the Clojure keywords from the `has/3` predicate (cf. Section 2.1.2). These functions merely have to instantiate and subsequently perform Cha-Q changes for each transformation subject.

However, we advocate delaying and grouping those changes in a `be.ac.chaq.change.ChangeSet`. Delaying changes enables development tools to simulate their eventual effect. For instance, to have end-users select the most appropriate code transformation from several possible ones. Implementation-wise, each function can extend the current change set for the compilation unit (i.e., the root of the AST) in which a

```java
public class Department {
    private String name;
    private LinkedList<Room> rooms;
    private Room wroom;
    private SPriorityQueue waitingroom;
    private LinkedList<Device> equipment;

    public Department(String n) {
        name = n;
        rooms = new LinkedList<Room>();
        wroom = new Room(0);
        waitingroom = new SPriorityQueue();
        equipment = new LinkedList<Device>();
    }

    public void addRoom(Room room) {
        rooms.addFirst(room);
    }

    public void removeRoom(Room room) {
        rooms.remove(room);
    }

    public Room findRoom(Number id) {
        for (int i = 0; i < getNoOfRooms(); i++) {
            Room room = (Room) rooms.get(i);
            if (room.getId() == id) return room;
        }
        return null;
    }
}
```

```java
public class Department {
    private String name;
    private HashMap<Number, Room> rooms;
    private Room wroom;
    private SPriorityQueue waitingroom;
    private LinkedList<Device> equipment;

    public Department(String n) {
        name = n;
        rooms = new HashMap<Number, Room>();
        wroom = new Room(0);
        waitingroom = new SPriorityQueue();
        equipment = new LinkedList<Device>();
    }

    public void addRoom(Room room) {
        rooms.put(room.getId(), room);
    }
    public void removeRoom(Room room) {
        rooms.remove(room.getId());
    }

    public Room findRoom(Number id) {
        return (Room) rooms.get(id);
    }
}
```

Figure 2.1: Example changes to be repeated by a CHAQEKO/X transformation.

node resides or create a new one if none is present. After all changes have been recorded, they can be applied by calling a function `apply-and-reset-rewrites`.

Moreover, as discussed in Section 1.3 R2, our ideas concerning templates for matching code can be transposed to templates for generating code. Instead of matching directives, rewriting directives `[<template-component>]@[<directive-name`$_1$`>` `...<directive-name`$_n$`>]` can be associated with the components of such a template. Each rewriting directive can be expanded into a functional expression that invokes the CHA-Q change API.

To link both sides of a specification together, we proposed a `(chaqeko/x <LHS> <RHS>)` macro. This macro is to perform the changes specified by its second argument `<RHS>` to all subjects that match its first argument `<LHS>`. For instance, by expanding into the appropriate Clojure `for`-expression. At the same time, this macro can automate declaring meta-variables.

The following transformation specification illustrates the resulting CHAQEKO/X language on a bigger example:

```
1  (chaqeko/x
2   [public class ?class {
3      [[private LinkedList<?type> ?list;]@[(equals ?fielddeclaration)]
4      [public ?constructor() {
5       [[?listref2]@[(refers ?list)]=new LinkedList<?type>();]@(equals ?init)
6      }]@[contains]
7
8      [public void ?add(?type ?param1) {
9        [?listref3]@(refers ?list).addFirst([?arg3]@(refers ?param1));
10     }]@(equals ?insert)
11
12     [public ?type4 ?find(?argType ?param5) {
13      [[?el]@(expression-type ?type).?getKey()]@(parent+)
14     }]@(equals ?lookup)]
15  }]@[contains]
16  =>
17  [private HashMap<?argType,?type> ?list;]@(replace ?fielddeclaration)
18  [?listref2=new HashMap<?argType,?type>();]@(replace ?init)
19  [public void ?add(?type ?arg1) {
20    ?listref3.put(?param1.?getKey(),?param1());
21  }]@(replace ?insert)
22  [public ?type4 ?find(?argType ?param5) {
23    return (?type4)?list2.get(?param5);
24  }]@(replace ?lookup)
25 )
```

The transformation repeats changes similar to the ones depicted in Figure 2.1. They convert a field of type list into a keyed collection. Lines 3–6 and 17–18 convert the `?fielddeclaration` and its associated initialization statement `?init` in the constructor of `?class`. Note how type parameters are provided for the new `HashMap` instance. Lines 8–10 and 19–21 convert method `?add` from using `List<Value>.addFirst(Value)` to using `HashMap<Key,Value>.put(Key, Value)`. To this end, line 13 retrieves the type of and the method that can be used to compute a value's key.

3 Conclusion

We have introduced ChaQeko/X as the transformation specification formalism of the Cha-Q project. As in most approaches to program transformation, our 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 the identified subjects should be rewritten.

ChaQeko/X is a template-based transformation language. On the LHS of a transformation, code templates function as a means to specify the structural (i.e., instructions and their organisation) and behavioural characteristics (i.e., the order in which instructions are executed at run-time and the values instructions operate on) of the stransformations subjects. We owe this specification style to the Soul meta-programming language [17, 18], but innovate by offering a suite of operators that can be applied to a template to change the way it is matched —thus paving the way for future operator-based tool support for applying transformations in a change-aware manner (cf. Deliverable 3.2.b). On the RHS of a transformation, code templates function as convenient short-hand for functional expressions that instantiate and perform first-class Cha-Q changes on the transformation's subjects.

4 References

[1] E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. Special issue on Reduction Strategies in Rewriting and Programming. [5]

[2] E. Visser. Program transformation with stratego/xt. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 315–349. Springer Berlin / Heidelberg, 2004. [5]

[3] M. van den Brand, J. Heering, P. Klint, and P. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002. [5]

[4] M. Appeltauer and G. Kniesel. Towards concrete syntax patterns for logic-based transformation rules. In *Proceedings of the Eighth International Workshop on Rule-Based Programming (RULE07)*, 2007. [5]

[5] 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. [5, 6]

[6] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In *Proceedings of the 10th International Conference on Compiler Construction (CC01)*, pages 52–68, 2001. [5]

[7] 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)*, pages 133–144, 2002. [5]

[8] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In Dieter Rombach and Mary Lou Soffa, editors, *Proceedings of the 28th International Conference on Software Engineering (ICSE06)*, pages 172–181, 2006. [6]

[9] H. Falconer, P. Kelly, D. Ingram, M. Mellor, T. Field, and O. Beckmann. A declarative framework for analysis and optimization. In *Proceedings of the 16th International Conference on Compiler Construction (CC07)*, 2007. [6]

[10] T. Cohen, J. Gil, and I. Maman. Guarded program transformations using jtl. In *Proceedings of the 46th International Conference on Objects, Models, Components and Patterns (TOOLS08)*, 2008. [6]

[11] Y. Padioleau, J. Lawall, and G. Muller. SmPL: A domain-specific language for spec-

ifying collateral evolutions in linux device drivers. *Electronic Notes in Theoretical Computer Science - Proceedings of the 2006 International ERCIM Workshop on Software Evolution*, 166:47–62, 2006. [6]

[12] Y. Padioleau, J. Lawall, R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys08)*, 2008. [6]

[13] J.. Lawall, J. Brunel, R. Hansen, H. Stuart, and G. Muller. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN09)*, 2009. [6]

[14] Coen De Roover and Reinout 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. [9, 13]

[15] Siltvani. A workbench for template-driven program transformation. Master's thesis, Vrije Universiteit Brussel, 2013. [9, 10, 13, 16]

[16] William E. Byrd. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. PhD thesis, Indiana University, August 2009. [9, 14]

[17] Coen De Roover, Carlos Noguera, Andy Kellens, and Viviane 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. [10, 16, 23]

[18] Coen De Roover. A logic meta-programming foundation for example-driven pattern detection in object-oriented programs. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM11)*, 2011. [10, 19, 23]

[19] Coen De Roover, Verónica Uquillas Gómez, Alessandro Murgia, and Javier Pérez. Chaq deliverable 2.1a: Definition of a meta-model for representing changes in software. Technical report, Vrije Universiteit Brussel, 2013. [13]