NICE: A Flexible Expression Language

Oliver Hacker[1] and Thomas Buchmann[2]
[1]*Chair for Applied Computer Science I, University of Bayreuth, Universitätsstrasse 30, 95440 Bayreuth, Germany*
[2]*Faculty for Computer Science, Deggendorf Institute of Technology, Dieter-Görlitz-Platz 1, 94469 Deggendorf, Germany*

Keywords: Model-Driven Development, Domain-Specific Languages, Expressions, Reuse, Code Generation.

Abstract: Model-driven development relies on model-transformation languages to describe mappings between different metamodels and as such facilitate a model-first workflow. These languages and accompanying tools have matured a lot over the past years. However, the more recent developments BXtend and BXtendDSL lack some features like an integrated expression language which would greatly improve the usability. In this paper, we present the flexible expression language NICE, which aims to solve the aforementioned problem in a modular, reusable, and adaptable way, while being fast and easy to use.

1 INTRODUCTION

Model-driven software engineering (MDSE) (Frankel, 2003; Völter et al., 2006) puts strong emphasis on the development of high-level models rather than on source code. Models are not considered documentation or informal guidelines on how to program the actual system. In contrast, models have a well-defined syntax and semantics. Moreover, model-driven software engineering aims at raising the level of abstraction by the development of *executable* models. *Code generators* are used in model-driven software engineering to transform the specification of higher-level models into source code.

Among others, *domain-specific languages (DSLs)* and accompanying code generators are used to achieve the goal of MDSE. A *domain-specific language (DSL)* is a programming or specification language which is dedicated to a particular problem domain. The vocabulary of the language is based on abstractions which are closely aligned with the domain for which the language is built (Voelter et al., 2013). Typically, DSLs come with a syntax which allows to express these abstractions in a concise way, either by text, symbols, graphics, tables or any combination thereof.

In terms of MDSE, the central aspect of a DSL is the underlying metamodel. *Metamodels* are used to define the core concepts (abstract syntax) of modeling languages. For object oriented modeling, the Object Management Group (OMG) provides the *Meta Object Facility (MOF)* standard (OMG, 2015). The MOF standard has been adopted widely for defining metamodels. A subset of MOF has been implemented in the *Eclipse Modeling Framework (EMF)* (Steinberg et al., 2009) and its meta-metamodel *Ecore*.

Over the years a complete ecosystem of model-driven languages and accompanying tools has emerged on top of EMF, including frameworks for the specification of domain-specific languages. *Xtext* (Bettini, 2016) is dedicated to the development of programming languages and domain-specific languages. As a language workbench, it provides support for all aspects of a language infrastructure, ranging from parsing, over linking to code generation or interpretation. The framework is fully integrated into the Eclipse IDE and its internal data structures are based on Ecore.

In this paper, we present NICE (**N**ew **I**deas for **C**ollection **E**xpressions) – a flexible expression language including a code generator for Java which may be integrated into Xtext-based domain-specific languages. As a proof of concept, we demonstrate the integration of NICE into our model transformation language BXtendDSL (Buchmann et al., 2022).

The paper is structured as follows: In section 2, related work is discussed and a background to our work is provided. Our chosen approach is detailed in section 3. An evaluation of our language is given in section 4, while section 5 concludes the paper.

Hacker, O. and Buchmann, T.
NICE: A Flexible Expression Language.
DOI: 10.5220/0011712700003402
In *Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2023)*, pages 63-74
ISBN: 978-989-758-633-0; ISSN: 2184-4348

2 BACKGROUND AND RELATED WORK

Over the years, many approaches and accompanying tools for model transformations have been proposed, which all allow for declarative specifications of the respective model transformation problem. However, our own observations and case studies (Anjorin et al., 2020; Bank et al., 2020; Westfechtel and Buchmann, 2018; Buchmann and Westfechtel, 2016; Greiner and Buchmann, 2016; Buchmann and Greiner, 2016b; Buchmann and Greiner, 2016a) have revealed that all of the approaches have limitations when conditional creation of target elements is required. In this case, imperative approaches like BXtend or BXtendDSL are more powerful.

If we take a closer look at existing domain-specific (model transformation) languages in the EMF ecosystem, e.g. QVT-R (OMG, 2016), ATL (Jouault et al., 2008), Acceleo[1], or EVL+Strace (Samimi-Dehkordi et al., 2018) it is obvious, that each of those languages uses expressions to navigate along model elements and to express constraints on model elements, like e.g., guards (ATL and Acceleo) or pre- and postconditions of transformations (when and where clauses in QVT-R). While the first three languages rely on OCL (OMG, 2014), EVL+Strace makes use of the Epsilon Object Language (EOL) (Kolovos et al., 2006), which is also based on OCL expressions.

Since there is evidence (Hebig et al., 2018) that functional-style languages like OCL and its derivatives are harder to comprehend than their imperative counterparts, a different approach to embedded expression languages is beneficial in that regard. This is further supported by the fact that OCL was never intended to describe behavior (only state and consistency) which requires extensions and modifications not compatible with the OCL standard (Cuadrado et al., 2008; Brucker et al., 2014; Jouault and Beaudoux, 2015; Jouault et al., 2015).

Furthermore, all of the above approaches that incorporate OCL or adapted variants, rely on an interpreter to execute the model transformations. However, since we intend to use our expression language in BXtendDSL, code generation is required for interoperability. Due to the problems laid out in the previous paragraph, a code generator for OCL would also not solve these issues.

Therefore, we set out to develop our own expression language, NICE, using the common DSL framework Xtext[2]. While Xtext already includes an embeddable language fragment, called Xbase (Efftinge et al., 2012), that can in principle be added to any Xtext-based language, we decided not to built upon it in our use case for several reasons: First, Xbase is heavily inspired by Java's syntax and behavior, making it more like a general purpose language. While it is not impossible to add custom language constructs, the preexisting infrastructure (provided for fast prototyping) has to be heavily modified, which is difficult without knowing the intricacies of Xbase's implementation due to a lack of documentation of many internal components. Second, our goal is to simplify common tasks necessary when developing model transformations using our expression language. Using Xbase as a basis however would not be much different from the existing approach of specifying transformation logic in Xtend files (since Xbase is a part of Xtend). Since a new language always requires a familiarization phase, we can leverage this time to teach a user how to express their transformation concisely and efficiently using the specialized features provided by the language.

[1] https://www.eclipse.org/acceleo/

[2] http://www.eclipse.org/Xtext

3 APPROACH

The following chapter discusses in detail the design decisions behind our language, its implementation using the Xtext framework and the code generation engine. We designed the first version of our language with the intent to use it in our model transformation language BXtendDSL.

3.1 Design Decisions

When taking a closer look at the usage of BXtendDSL in several different transformation scenarios provided with the BenchmarX (Anjorin et al., 2017b) testcases, some common traits can be observed:

Acyclic Control Flow. The control flow leads from the input parameters to the return value just by using conditions and function calls. Recursion and `while` loops are only used in rare cases.

Imperative Nature. The transformation of input objects into output objects is specified in a purely imperative way. Object-oriented concepts are not employed. This is due to the fact that BXtendDSL uses the *generation gap* pattern (Fowler and Parsons, 2010; Fowler, 2010) and hook methods, which simply do not allow using many of the object-oriented concepts.

Collections. Processing collections is a central aspect in many transformation languages, e.g., when

dealing with multivalued features or groups of objects. In this case, the Xtend language proves useful due to the support for collection literals and `foreach` loops. Furthermore, the developer is provided with a *fluent interface* (Fowler and Parsons, 2010; Fowler, 2005) following the method-chaining pattern utilizing the extension method mechanism and a sophisticated standard library (Mikosik, 2021).

No Side Effects. Most transformations may be realized just with information obtained from the input parameters and their respective correspondence relations. But there are some exceptions, like the Families2Persons-Benchmark (Anjorin et al., 2017a), which requires additional relations between model elements to avoid information loss.

Null Safety. Especially in the case of incremental transformations, null values may occur an thus have to be handled easily. To this end, Xtend provides the Elvis-operator "`?:`", which evaluates the second argument if the first one is null, and also the safe-navigation-operator "`?.`", which evaluates the whole expression to null if the object to the left of the `?` is null. In case a feature of an input object has a null value or the input object itself is null, this state shall be propagated to the output object. Still, a way to check nullity is required for assigning standard values to output objects instead.

Based on these observations, similarities to existing applications are obvious. The first three items are core properties of pipeline-based architectures, which are designed to process a sequence of objects in many smaller steps. The last two items may be identified as typical properties of functional programming languages. A strictly functional language however is not the best choice, since side effects are required in some model transformations.

As a consequence, our expression language borrows some constructs from the fields above. For instance, the basic structure of our textual language is based on a generalized pipeline model, which also allows for branching in the object flow. Figure 1 depicts a simplified diagram of this idea. The input nodes are shown in green color at the very top, followed by three processing layers which finally end up in the output layer depicted in red color. Please note that nodes in a processing layer may depend only on results of the previous layers, leading to an acyclic graph.

While the straight-forward approach would be to convert this diagram into graphical syntax, we opted for a textual syntax instead. For one, because this makes it easier to integrate the expression language with existing textual languages. Additionally, tools

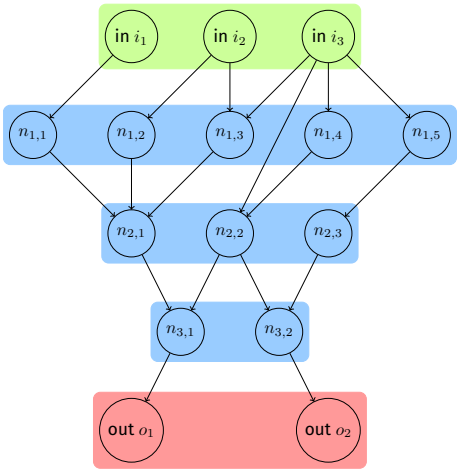

Figure 1: Possible object flow in our expression language.

for textual DSLs are much more mature and well maintained.

Listing 1 shows the object flow depicted in figure 1 in the textual syntax of our expression language. Input nodes are denoted using the keyword `in`. Links between nodes are created by referencing them using their names.

```
1 in -> i1, in -> i2, in -> i3
2 | fun n11: use(i1), fun n12: use(i2), fun n13: use(i2
      , i3), fun n14: use(i3), fun n15: use(i3)
3 | fun n21: use(n11, n12, n13), fun n22: use(i3, n14),
      fun n23: use(n15)
4 | fun n31: use(n21, n22), fun n32: use(n22, n23)
5 | out(n31) -> o1, out(n32) -> o2
```

Listing 1: Textual representation of figure 1.

The left hand side of the arrow `->` contains an (optional) argument which is bound to the input node. It may contain all elements on which the navigation operator `.` may be used, like e.g., variables, method calls or classes. Since the expression language will be used in BXtendDSL, arguments of input nodes may be features of source or target elements (cf. listing 2, l. 5).

Output nodes are denoted using the keyword `out`. Objects, which are the results of nodes listed in the brackets are assigned to an output element. Note, that listing multiple nodes in a single output node results in aggregating the incoming objects into a single collection. The output element itself is optional and is valid within the surrounding scope. If there is no output element, a return value will be created (cf. listing 2, l. 6).

Transformation nodes are composed of a single expression or the keyword `pass`. This type of node is used to transform an incoming stream of objects into an outgoing stream of objects. Since functions, introduced by the keyword `fun`, are treated as a single

expression, transformation nodes may become arbitrarily complex (cf. listing 2, l. 7).

Nodes are aggregated into pipelines which model the object flow. Pipelines may be nested inside blocks of statements and thus act similarly to `foreach` loops known from imperative programming languages (cf. listing 2, ll. 1 and 9).

The inner structure of the expression language is strongly influenced by procedural and functional aspects. Furthermore its syntax is inspired by the programming language Python (as it is also white-space aware).

Statements as the most general language construct are divided into pipelines (as mentioned above), variable declarations, control flow manipulation, and expressions. Since variables are meant to always be final to avoid problems with captures, the right hand side of variable declarations may contain arbitrary expressions and pipelines, allowing for control structures directly in the initialization. The keywords `yield` and `break` manipulate the control flow, which may also return optional values for the surrounding block. In our language, `yield` is a mixture between Java's `return` statement and Python's `yield` statement and thus it resembles the yield expression used in `switch` statements, introduced in Java 14. The keyword `pass`, also known from Python, is used to fill empty sequences of statements to allow for white-space awareness of the language (cf. listing 2, l. 9).

Finally, arbitrary expressions are grouped into typical control structures, binary and unary operators, as well as navigable elements (cf. listing 2, ll. 10-22).

```
1  Pipeline: Level ('|' Level)+
2  Level: Node (',' Node)*
3
4  Node: InputNode | OuputNode | TransformationNode
5  InputNode: 'in' Accessible? ('->' ID)?
6  OuputNode: 'out' ('(' ID ('+' ID)* ')')? ('->'
       (Accessible | 'void'))?
7  TransformationNode: Expression | 'pass'
8
9  Statement: Pipeline | VariableDeclaration |
       Expression | 'yield' Expression? | 'break'
       Expression? | 'pass'
10 Expression: Function | If | Switch | While |
       BinaryOperatorCall
11 Function: 'fun' ID? ('(' Variable (',' Variable)*
       ')')? ':' (Expression | Statements+)
12 BinaryOperatorCall: PrefixOperatorCall BinaryOperator
       PrefixOperatorCall
13 PrefixOperatorCall: PrefixOperator?
       PostfixOperatorCall
14 PostfixOperatorCall: Accessible PostfixOperator?
15 Accessible: (Literal | '(' Expression ')') Access*
16 Access: CallOperator | '.' ID | '::type' | '::meta' |
       '::raw'
17 CallOperator: '(' (Expression (',' Expression)*)? ')'
18
19 Literal: Primitive | Collection | IdReference
20 Primitive: IntLiteral | FloatLiteral | StringLiteral
       | BoolLiteral
21 Collection: List | Set | Map
```

Listing 2: Excerpts from NICE's grammar in an Xtext-like EBNF notation. Rules not shown are defined similarly to Python's syntax.

3.2 Implementation

In this section we give an overview of the implementation of the expression language.

3.2.1 Grammar

We use the Xtext framework to implement our expression language. Xtext allows to specify a context free grammar enriched with information on how to populate the AST, which is based on an Ecore model (Steinberg et al., 2009). An ANTLR3 grammar is automatically derived from the Xtext grammar and used for parser generation. A few modifications in Xtext's parser handling have been made to facilitate more flexible white-space awareness and less rigid operator definitions.

3.2.2 Java and EMF Interoperability

A very important feature for our expression language is interoperability with one (or more) high level programming languages. Since Eclipse and Xtext constitute our environment, Java is the natural choice. Furthermore, Xtend/Xbase are languages created with the Xtext framework which already provide such an interoperability allowing us to built upon existing systems. In particular we can reuse the provided Java metamodel and accompanying tools like type references.

The main goal is to support Java types and Ecore types in our expression language by using a scope provider which creates a model for Java packages using the classpath of the respective Eclipse project. In the current version of our expression language, we do not support an import mechanism for classes, since we intend to use the expression language as a sub-language for our model transformation language, which already supports import mechanisms. Based on classes, static methods and constructors may be called resulting in complete interoperability.

Another benefit is our standard library. We follow a modular approach by providing the annotations `@NICEDefaultImportProvider` and `@NICEDefaultImport`. While the expression language itself does not support imports on a file

level, these annotations allow for specifying project-specific standard imports. Both annotations influence the behavior of our scope provider.

An application of this default import mechanism can be found in the realization of the language's operators. For this purpose, static methods with corresponding names are annotated with the respective import annotations and imported into the global scope. This mechanism allows for the redeclaration of operators similar to C++ or Python, since local definitions overwrite existing definitions in the classpath.

Furthermore, operators may be decorated with the Xbase annotation `@Inline` which uses the String contained in the annotation instead of a method call.

Our language does not only allow for the adaption of existing operators using imports, but also for declaring additional operators. To this end, the class `NICEDEfaultOperatorDeclarations` has to be extended. An operator in the expression language consists of a character sequence, a name for the implementing method(s), an arity for the operator and a precedence. In case of operator overloading, the operator with matching parameter types is called.

Since operator precedences may be assigned arbitrarily, they are not hard-coded into the grammar as usual, but must be corrected explicitly by post-processing the AST.

3.2.3 Type System and Type Inference

Xbase's Java metamodel and its accompanying helpers mentioned above are also used for typing our expression language. NICE does not support explicit typing by design, instead types of composite expressions are inferred. This further helps the Python-esque appearance and usage of our language.

Since some of the tools for Xbase's type inference work just within the bounds of its Java metamodel (and are thus independent of Xbase itself), we can build upon them. Unfortunately the type inference mechanism per se is not implemented in a language-independent way and thus we had to reimplement it for NICE.

NICE's type inference is realized by an external visitor, which traverses language model elements. In case a method or operator which is not yet linked is found, the possible scope for this context is used to perform overload resolution and preemptive linking (which is queried by the scope provider at a later time). When visiting EMF elements, like `EOperation` or `EStructuralFeature`, the respective type is mapped to the Java metamodel in order to allow usage of the helper class `LightweightTypeReference`, which is part of Xbase's type computation system and contains many useful functionalities. The standard behavior for resolving EMF elements is to check the respective generator model and find the generated Java classes. If that is not possible, e. g. because we can't find the generator model, a workable substitute is built on demand.

3.3 Semantics and Code Generation

In order to execute NICE programs, we provide a code generator producing readable Java code. In contrast to using an interpreter, native Java code provides performance benefits and an easier integration into existing Java projects. In the following subsections, we will briefly describe the respective parts of the code generator and in the process the semantics of the expression language.

3.3.1 Mapping of Pipelines

Each NICE program starts with a top level element consisting of exactly one pipeline which contains the remaining elements. As described in section 3.1, the first layer only contains input nodes whereas the last layer only contains output nodes. Input and output nodes, which are declared on intermediate layers are treated for code generation purposes as if they were part of the input or output layer respectively.

The outermost pipeline is transformed into a Java class which contains Java code generated from its inner elements. Depending on the number and type of output nodes, the generated Java class implements the respective functional interfaces in Java's (`java.util.function`) and Xbase's (`org.eclipse.xtext.xbase.lib.(Functions|Procedures)`) standard libraries.

Nested pipelines, which are declared within a parent pipeline are also mapped to Java classes. Since Java allows nesting classes, the pipeline structure may be directly reflected.

Figure 2 depicts a root pipeline with the properties described above and the generated components from the respective input and output nodes, as described in the following subsections.

Pipelines, which reflect a linear object flow with only one input and output node, and on each intermediate layer only one transformation node respectively, are mapped directly to a corresponding chain of Java Stream methods, following the method-chaining pattern.

3.3.2 Mapping of Input Nodes

Input nodes contribute three artefacts for the generated pipeline class: (1) parameters of the constructor

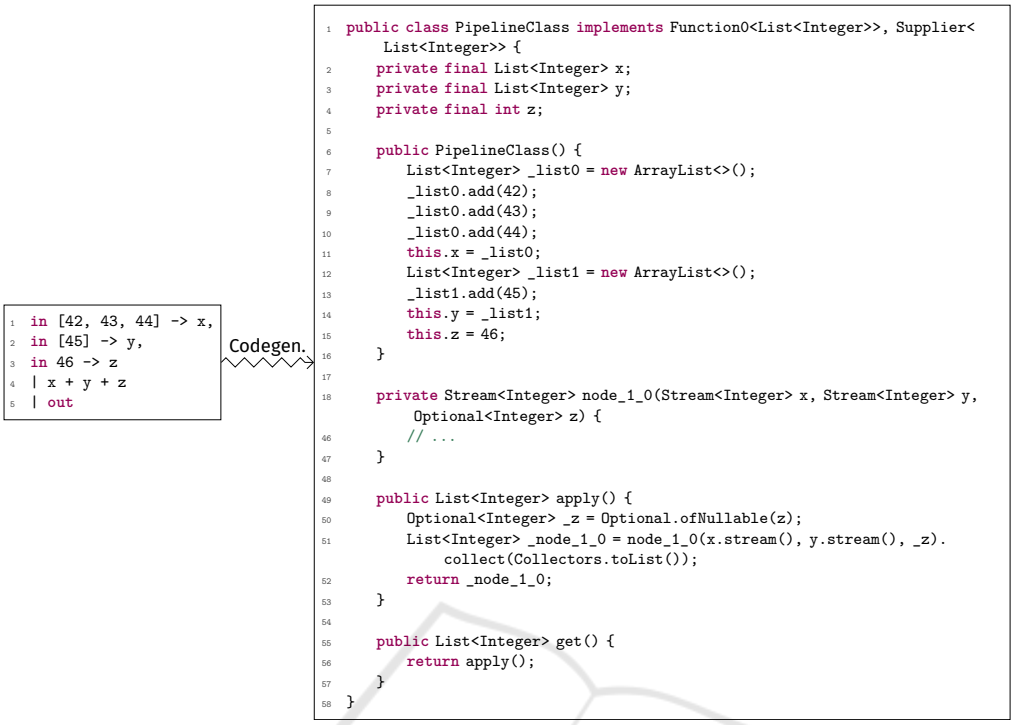

Figure 2: Pipeline generation. The method `node_1_0` is omitted due to its complexity and length.

are derived from external dependencies of the node (if present). (2) The argument expression on the left hand side of the `->` is transformed into statements which are placed into the body of the constructor of the generated class. (3) A field with the corresponding assignment in the constructor is generated, which has the type of the argument expression. This allows for arbitrary preprocessing steps in the argument of the input node, or assertion of preconditions as long as they consist of a single expression.

3.3.3 Mapping of Output Nodes

Processing the output nodes is done in the context of the last layer, which from the code generator's perspective just contains all output nodes. The code generator for this last layer creates execution methods, which implement the interfaces of the pipeline class. Java elements generated from output nodes are placed within those methods. In addition, the remaining single nodes are linked here according to the following steps: fields of classes generated from input nodes are transformed into null-safe objects (Stream/Optional) and methods generated from transformation nodes are called with respect to their dependencies. Finally, statements generated from output nodes are inserted, returning objects processed by the pipeline.

We have to distinguish between several cases when generating code for output nodes: In case there is only a single output node, which does not contain an output element, a pipeline with return value, whose type depends on the single output node is generated. If the output element `void` is specified explicitly this results in a pipeline without return value. For multiple output nodes without explicit output elements, a helper class for the return value is created containing the respective fields. Output nodes with output elements, result in generated parameters for the method to be executed, requiring their value to be settable via side-effects.

In case the output element is a variable, an assignment is generated. Since Java does not support pointers to variables, this case always has to be treated from the surrounding context of the pipeline, i.e. during the compilation of the output node this case is treated as if no output element was specified, which results in the creation of a return value as described above.

A feature access as an output element allows for a local decision, since the context – the object which is accessed – is present as a method parameter. In case of a multi-valued feature, we check for a getter which allows for calling the methods `clear` followed by `add` or `addAll`. If this is not the case, we search for a setter – like in case of a single-valued feature. If a setter is also not present, the respective fields are checked for visibility and a possible final keyword in order to generate an assignment.

3.3.4 Mapping of Transformation Nodes

As stated above, transformation nodes are mapped onto methods. Dependencies to nodes on preceeding layers are passed as parameters. All nodes in NICE are either of type `Optional` or when multiple elements are involved of type `Stream` to use the capabilities of both APIs and to provide null-safety as far as possible. The methods generated from transformation nodes are also just containers for statements which result from the contained NICE expression. We also can distinguish three cases here:

Mapping Nodes. A mapping node is a simple transformation node which only contains a single, multi-valued dependency, i.e. it requires only one parameter of type `Stream` and does not access it in the context of the special `raw`-access. Consequently, references to that node act in NICE just as references to single element of the underlying stream. Thus, the method `Stream::map` may be called on the single parameter, while the remaining logic is contained in the passed lambda expression.

Filtering-If Nodes. A so called *filtering-if node* is present if a node fulfills all properties of a mapping node and in addition contains a single `if`-statement or a function with an `if`-statement as last statement, which has the following properties:

- the last statement is either an `if` or `yield` statement.
- statements prior to the last statement can not terminate the control flow preemptively (i.e., they don't contain `yield` statements).
- the conditions above hold recursively for the last statements in the body of the `if` statement.

The advantage when using filtering-if is that this construct can be compiled directly to a call of the method `Stream::filter`.

Aggregating Nodes. Nodes with multiple dependencies of type `Stream` result in the generation of an aggregating node. Since referenced nodes in the expression language contain their component type, the generated code must allow access to single elements of the incoming `Stream`s. This is achieved by a generated implementation of the `Spliterator` interface.

3.3.5 Mapping of Statements

A characteristic of NICE statements is the option to be grouped into blocks with further statements. Statements subsume all language constructs in the expression language, like pipelines, variable declarations, expressions, as well as `yield`, `pass` and `break` statements. Pipelines used as statements have already been discussed in section 3.3.1. Variable declarations result in the generation of a local variable declaration in Java with the respective type. `break` statements may also be mapped directly to the corresponding Java statement, as they may only occur in `while` loops and `switch` statements. Treating `yield` statements is very similar: If they occur in a pure statement context, i.e. expressions in the containment hierarchy of the surrounding function or node are used as statements, `return` statements are generated. Otherwise they function like an assignment to an expression's result variable. Finally, the `pass` statement also has a different semantics depending on the context where it is used. If it is used in a list of statements, `pass` acts like an empty statement `;` in Java. The pass statement may also be used instead of the implementation of a transformation node. In this case, the result of the node with the same index of the previous layer is passed through.

3.3.6 Mapping of Expressions

When mapping expressions, we have to consider two different categories: NICE expressions that may be directly transformed into Java expressions and those that require multiple Java statements. A compiled NICE expression always consists of an arbitrary number of statements and a single expression which can be consumed by further statements. This allows for an easy recursive mapping of nested NICE expressions which result in multiple Java statements with the ability to return a resulting expression for the outermost element.

Operators are directly compiled into the respective method calls of the linked operator methods.

Transforming primitive literals is also straightforward, since NICE only supports `int`, `double`, `String` and `boolean`. They are mapped onto the respective Java literals.

Functions may also be generated in an easy way, since the most complex part comprises the implementation of the type inference, as stated above. If functions are used as arguments of method calls, they are compiled into the respective lambda expressions. In case functions are used as the content of a transformation node, this is not possible. Instead, they are directly generated into the method body of the method generated for the transformation node. This allows nodes to consist of multiple statements.

Since Java does not support collection literals, they must be split into multiple statements. To not restrain users in using them, no helper methods like `List.of(...)` or `Map.of(...)` are used, as they

result in unmodifiable collections. We follow a very pragmatic approach instead: Lists result in instances of `ArrayList`, Sets in `LinkedHashSet` and dictionaries in `LinkedHashMap`. This allows for preserving the ordering of the declared elements. After an instance of a collection has been created, elements are added one after another using the respective methods.

An access expression always results in the creation of a respective access in Java, which may differ significantly from the NICE expression depending on the type of navigation operation and linked feature.

Generation of named references depends on the referenced object. For elements of the expression language and external parameters required for interoperability with BXtendDSL, the surrounding scope contains a Java variable which will be linked. If named references are used to access members, we distinguish between EMF and Java elements, the first of which is mapped to usable Java expressions using the EMF generator model, while the latter ones can simply be inserted.

The control structures `while`, `if`, and `switch` are mapped to their corresponding Java counterpart. Since they may occur as right hand sides of expressions, a corresponding variable holding the result of the expression is created when necessary.

4 EVALUATION

As a proof of concept, we integrated the expression language NICE into our model transformation language BXtendDSL (Buchmann et al., 2022; Bank et al., 2021) in order to express a larger part of a model transformation on the declarative layer.

Since Xtext provides built-in mechanisms for reuse and composition of existing grammars, using NICE from within the BXtendDSL grammar is easy. Please note that slight changes in the syntax of BXtendDSL were introduced when we integrated NICE. Especially, the filter modifiers of BXtendDSL are no longer referred to using the pipe (`|`) symbol to avoid conflicts with NICE. Instead, BXtendDSL filters are now specified using the ”greater” (`>`) symbol. Furthermore, the mapping operators (`<-->`, `-->`, and `<--`) have been replaced with named blocks (`oneway mapping` and `mapping`).

4.1 Transformation Problem

In order to allow for a detailed comparison, we discuss the benefits of NICE using the *Families-to-Persons* transformation example (Anjorin et al., 2017a). A detailed description of the transformation problem and the solution using BXtendDSL (without the expression language) is given in (Bank et al., 2021). Due to space restrictions, the reader is kindly referred to (Bank et al., 2021) for more details.

4.2 Solution

For solving the *Families-to-Persons* transformation problem using BXtendDSL and NICE, we implemented feature mappings and modifiers directly on the DSL layer. Since NICE expressions are compiled into source code, this results in less specification effort required on the imperative layer of BXtendDSL. In contrast to the BXtendDSL specification without NICE discussed in (Bank et al., 2021), the classes `Member2FemaleUserImpl` and `Member2MaleUserImpl` are no longer necessary, since the corresponding `filter` modifiers are now directly expressed using NICE, as shown in lines 67-72 and 79-84 of listing 3.

Furthermore, the hook methods in the forward direction in class `Member2PersonUserImpl` is no longer required, since the respective behavior is now specified with NICE, as shown in lines 27-40 and 47-60 of listing 3.

This helps to dramatically reduce the user supplied code which was required on the imperative layer in the solution discussed in (Bank et al., 2021). However, the current solution using NICE still requires a small portion of user supplied code on the imperative layer: the backward mapping requires access to the transformation options specified in lines 7 and 8 of listing 3, which NICE doesn't yet support. Furthermore, the `create` modifier can't be compiled, since it sets a newly created person's birthday using a cache in the class `Member2PersonUserImpl`, which is not available in the pipeline (since a rule's instance is not easily accessible).

4.3 Empirical Results

In the following, we provide a quantitative analysis, a qualitative analysis as well as a performance analysis.

4.3.1 Quantitative Analysis

In our quantitative analysis, we compare the specification effort required to realize both solutions. Since in both cases, a textual concrete syntax is used, a quantitative impression of the size of the transformation definitions may be obtained by counting the number of *lines of code* (excluding empty lines and comments, as well as automatically generated lines), the *number of words* (character strings separated by whitespace)

```
trafo:
    sourcemodel "platform:/plugin/Families/model/Families.
        ecore"
    targetmodel "platform:/plugin/Persons/model/Persons.ecore"
    package "families2persons.inheritance.nice"

options:
    PREFER_CREATING_PARENT_TO_CHILD
    PREFER_EXISTING_FAMILY_TO_NEW

rule Register2Register:
    src:
        FamilyRegister s
    trg:
        PersonRegister t

abstract rule Member2Person:
    src:
        FamilyMember member
    trg:
        Person person > creation
    oneway mapping:
        member.name, member.motherInverse, member.fatherInverse,
        member.sonsInverse, member.daughtersInverse
        to
        person.name
        specifying forward:
            in member.name -> name, in member.motherInverse ->
                motherInverse,
            in member.fatherInverse -> fatherInverse, in member.
                sonsInverse -> sonsInverse,
            in member.daughtersInverse -> daughtersInverse
            | fun family:
                if motherInverse:
                    motherInverse
                else if daughtersInverse:
                    daughtersInverse
                else if fatherInverse:
                    fatherInverse
                else:
                    sonsInverse
            | family.getName() + ", " + name
            | out -> result.personName
    oneway mapping:
        member.motherInverse, member.fatherInverse,
        member.sonsInverse, member.daughtersInverse
        to
        person.personsInverse
        specifying forward:
            in member.motherInverse -> motherInverse, in member.
                fatherInverse -> fatherInverse,
            in member.sonsInverse -> sonsInverse, in member.
                daughtersInverse -> daughtersInverse
            | fun family:
                if motherInverse:
                    motherInverse
                else if daughtersInverse:
                    daughtersInverse
                else if fatherInverse:
                    fatherInverse
                else:
                    sonsInverse
            | fun familyRegister: family.getFamiliesInverse()
            | familyRegister::corr from Register2Register by t
            | out -> result.personsInverse
    oneway mapping:
        member.name from person.name with member

rule Member2Female follows Member2Person:
    src:
        FamilyMember member > filter:
            in member
            | if member.getDaughtersInverse() or member.
                getMotherInverse():
                true
            else:
                false
            | out
    trg:
        Female person

rule Member2Male follows Member2Person:
    src:
        FamilyMember member > filter:
            in member
            | if member.getSonsInverse() or member.
                getFatherInverse():
                true
            else:
                false
            | out
    trg:
        Male person
```

Listing 3: Solution to the Families-to-Persons transformation problem using BXtendDSL and NICE.

in these lines, and the *number of characters* in these words. Table 1 depicts the values obtained for those metrics for BXtendDSL and BXtendDSL+NICE. For both solutions, only those lines were counted which had to be written manually. In both cases, we counted the code specified directly on the DSL (declarative) layer, as well as the code required on the imperative layer. The same layout conventions and programming practices have been applied. As expected, moving logic from the imperative layer to the declarative one using NICE increases the size of the latter while shrinking the former by a similar amount. In total however, the size of the transformation definition increases slightly due to the introduction of new keywords and restructuring of BXtendDSL's grammar after integrating NICE, as explained above. The biggest benefit is the significant reduction of code required on the imperative layer, and thus, avoiding context switches for the transformation developer, since the transformation may now be described (almost) completely on the declarative layer.

Table 1: Amount of handwritten code of the transformation definitions of both solutions to the Families-to-Persons case, split by declarative and imperative layers.

Metric	BXtendDSL		BXtendDSL+NICE	
	Decl.	Imp.	Decl.	Imp.
Lines of code	24	65	83	37
#words	65	226	191	115
#characters	738	2512	1986	1728

4.3.2 Qualitative Analysis

In order to perform a qualitative analysis, test cases for the different transformation directions have been specified and executed for both batch and incremental mode of operation. We assume a test case to be passed, if the resulting model matches a predefined expected model state. The BXtendDSL solution is able to pass all tests specified in (Anjorin et al., 2020). The same holds for the BXtendDSL+NICE solution. Table 2 gives an overview of the tests and the obtained results following the criteria used in (Anjorin et al., 2020). Please note that two unexpected passes are due to cases that test for order-dependent update behavior (which state-based tools cannot provide).

The qualitative analysis shows that the correctness of the transformation is not affected by introducing the expression language NICE into BXtendDSL rules. Moreover, with respect to functional requirements both BXtendDSL and BXtendDSL+NICE achieve a perfect result: Both solutions pass all test cases. None

Table 2: Aggregate test results, grouped into categories and classified as expected/unexpected passes/fails.

Category	Result	BXtendDSL	BXtendDSL +NICE
Batch FWD	expected pass	7	7
	expected fail	0	0
	unexpected pass	0	0
	unexpected fail	0	0
Batch BWD	expected pass	11	11
	expected fail	0	0
	unexpected pass	0	0
	unexpected fail	0	0
Incr. FWD	expected pass	8	8
	expected fail	0	0
	unexpected pass	0	0
	unexpected fail	0	0
Incr. BWD	expected pass	7	7
	expected fail	0	0
	unexpected pass	1	1
	unexpected fail	0	0
Total	expected pass	33	33
	expected fail	0	0
	unexpected pass	1	1
	unexpected fail	0	0

of the other solutions compared in (Anjorin et al., 2020) exhibit a pass rate of 100%.

4.3.3 Performance Analysis

In order to evaluate the efficiency and scalability of the resulting transformation with respect to increasing model size, two experiments were conducted in both forward and backward directions for each of the transformation problems resulting in four sets of measurements: (1) batch transformations in forward and backward directions, and (2) incremental transformations in forward and backward directions. The batch transformations test how the solutions scale when creating corresponding opposite models of increasing size (model size up to 1 000 000 elements). For incremental transformations, the time required to locate and propagate corresponding changes to the dependent model is measured.

The tests were performed on the same machine and in isolation for each solution and each transformation problem. A desktop PC with an AMD Ryzen 7 3700X CPU was used, running at a standard clock of 3.60 GHz, with 32 GB of DDR4 RAM and Microsoft Windows 10 64-bit as the operating system. We used Java 13.0.2, Eclipse 4.11.0, and EMF version 2.17.0 to compile and execute the Java code for the scalability test suite. Each test was repeated 5 times and the median measured time was computed.

Our experiments show, that introducing an additional language on the declarative layer and a corresponding code generator introduces a slight overhead in runtime. But the overall transformation still resides in the same complexity class. While the times for forward and backward batch transformations are nearly the same for BXtendDSL and BXtendDSL+NICE, the pure BXtendDSL solution is slightly faster in both incremental cases.

The corresponding plots rendered from our experiments are shown in figure 3 and figure 4, respectively.

4.4 Summary

As expected, the use of BXtendDSL+NICE reduces the specification effort on the imperative layer significantly, by allowing to express large parts of the transformation now directly on the declarative layer. Functional correctness is not affected negatively; the full expressiveness of BXtendDSL is retained in the combination of BXtendDSL+NICE. Finally, the BXtendDSL+NICE solution proves scalable since it roughly exhibits linear performance. Compared to plain BXtendDSL, there are only small differences in computation time.

5 CONCLUSION

In this paper, we presented NICE – a flexible expression language including a Java code generator, which may be easily integrated into any Xtext-based domain-specific language. As a proof of concept, we integrated NICE into our model transformation language BXtendDSL. In the evaluation chapter in section 4 we show, that the language can be used to specify expressions in a declarative way and using the built in code generator, context switches in BXtendDSL between the declarative and the imperative layer are significantly reduced.

Future work deals with further improving the integration of NICE and BXtendDSL to also allow easy usage of BXtendDSL options, and to allow NICE pipelines to access elements declared on the imperative layer. Additionally, unnecessary repetitions in input layers of pipelines can be eliminated by inferring the available variables from the BXtendDSL context. To further reduce redundancy, stand-alone pipelines could be introduced to encapsulate common behavior. Another avenue to investigate is the addition of a graphical editor for NICE pipelines.

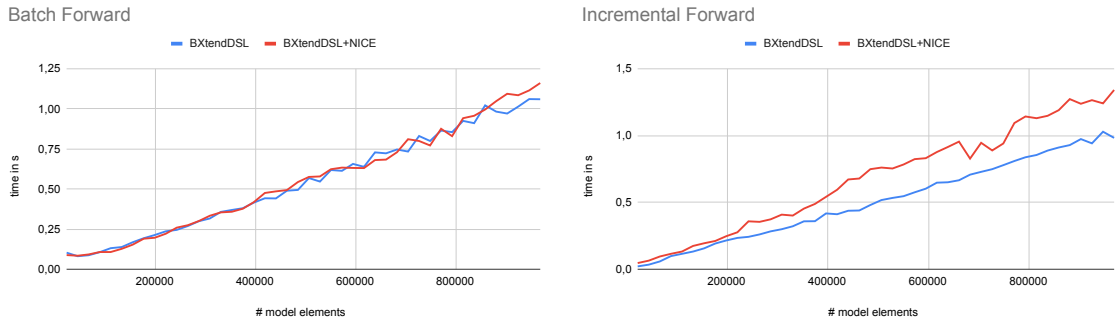

Figure 3: Forward batch (left) and incremental (right) transformation.

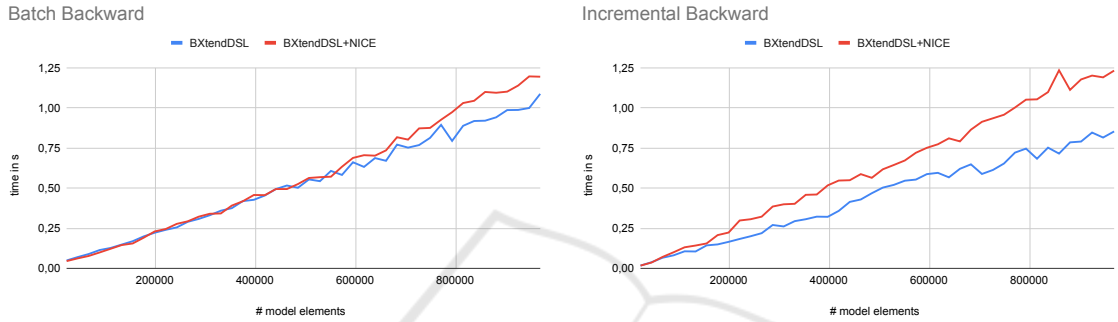

Figure 4: Backward batch (left) and incremental (right) transformation.

REFERENCES

Anjorin, A., Buchmann, T., and Westfechtel, B. (2017a). The families to persons case. In García-Domínguez, A., Hinkel, G., and Krikava, F., editors, *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017*, volume 2026 of *CEUR Workshop Proceedings*, pages 27–34. CEUR-WS.org.

Anjorin, A., Buchmann, T., Westfechtel, B., Diskin, Z., Ko, H., Eramo, R., Hinkel, G., Samimi-Dehkordi, L., and Zündorf, A. (2020). Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Softw. Syst. Model.*, 19(3):647–691.

Anjorin, A., Diskin, Z., Jouault, F., Ko, H., Leblebici, E., and Westfechtel, B. (2017b). Benchmarx reloaded: A practical benchmark framework for bidirectional transformations. In Eramo, R. and Johnson, M., editors, *Proceedings of the 6th International Workshop on Bidirectional Transformations co-located with The European Joint Conferences on Theory and Practice of Software, BX@ETAPS 2017, Uppsala, Sweden, April 29, 2017*, volume 1827 of *CEUR Workshop Proceedings*, pages 15–30. CEUR-WS.org.

Bank, M., Buchmann, T., and Westfechtel, B. (2021). Combining a declarative language and an imperative language for bidirectional incremental model transformations. In Hammoudi, S., Pires, L. F., Seidewitz, E., and Soley, R., editors, *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2021, Online Streaming, February 8-10, 2021*, pages 15–27. SCITEPRESS.

Bank, M., Kaske, S., Buchmann, T., and Westfechtel, B. (2020). Incremental bidirectional transformations: Evaluating declarative and imperative approaches using the ast2dag benchmark. In Ali, R., Kaindl, H., and Maciaszek, L. A., editors, *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020, Prague, Czech Republic, May 5-6, 2020*, pages 249–260. SCITEPRESS.

Bettini, L. (2016). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, Birmingham, UK.

Brucker, A. D., Clark, T., Dania, C., Georg, G., Gogolla, M., Jouault, F., Teniente, E., and Wolff, B. (2014). Panel discussion: Proposals for improving OCL. In Brucker, A. D., Dania, C., Georg, G., and Gogolla, M., editors, *Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014*, volume 1285 of *CEUR Workshop Proceedings*, pages 83–99. CEUR-WS.org.

Buchmann, T., Bank, M., and Westfechtel, B. (2022). Bxtenddsl: A layered framework for bidirectional model transformations combining a declarative and an imperative language. *J. Syst. Softw.*, 189:111288.

Buchmann, T. and Greiner, S. (2016a). Bidirectional model transformations using a handcrafted triple graph transformation system. In Cabello, E., Cardoso, J., Ludwig, A., Maciaszek, L. A., and van Sinderen, M., editors, *Software Technologies, 11th International Joint Conference, ICSOFT 2016, Lisbon, Portugal, July 24-26, 2016, Revised Selected Papers*, volume 743 of *Communications in Computer and Information Science*, pages 201–220. Springer.

Buchmann, T. and Greiner, S. (2016b). Handcrafting a triple graph transformation system to realize round-trip engineering between UML class models and java source code. In Maciaszek, L. A., Cardoso, J., Ludwig, A., van Sinderen, M., and Cabello, E., editors, *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT, Lisbon, Portugal, July 24 - 26, 2016*, pages 27–38. SciTePress.

Buchmann, T. and Westfechtel, B. (2016). Using triple graph grammars to realise incremental round-trip engineering. *IET Softw.*, 10(6):173–181.

Cuadrado, J. S., Jouault, F., Molina, J. G., and Bézivin, J. (2008). Optimization patterns for ocl-based model transformations. In Chaudron, M. R. V., editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, pages 273–284. Springer.

Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., and Hanus, M. (2012). Xbase: implementing domain-specific languages for java. In Ostermann, K. and Binder, W., editors, *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012*, pages 112–121. ACM.

Fowler, M. (2005). FluentInterface. https://www.martinfowler.com/bliki/FluentInterface.html.

Fowler, M. (2010). Generation gap. https://martinfowler.com/dslCatalog/generationGap.html.

Fowler, M. and Parsons, R. J. (2010). *Domain-Specific Languages*. Addison-Wesley Professional.

Frankel, D. S. (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley Publishing, Indianapolis, IN.

Greiner, S. and Buchmann, T. (2016). Round-trip engineering UML class models and java models: A real-world use case for bidirectional transformations with QVT-R. *Int. J. Inf. Syst. Model. Des.*, 7(3):72–92.

Hebig, R., Seidl, C., Berger, T., Pedersen, J. K., and Wasowski, A. (2018). Model transformation languages under a magnifying glass: a controlled experiment with xtend, ATL, and QVT. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming*, 72:31–39. Special Issue on Experimental Software and Toolkits (EST).

Jouault, F. and Beaudoux, O. (2015). On the use of active operations for incremental bidirectional evaluation of OCL. In Brucker, A. D., Egea, M., Gogolla, M., and Tuong, F., editors, *Proceedings of the 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015*, volume 1512 of *CEUR Workshop Proceedings*, pages 35–45. CEUR-WS.org.

Jouault, F., Beaudoux, O., Brun, M., Clavreul, M., and Savaton, G. (2015). Towards functional model transformations with OCL. In Kolovos, D. S. and Wimmer, M., editors, *Theory and Practice of Model Transformations - 8th International Conference, ICMT@STAF 2015, L'Aquila, Italy, July 20-21, 2015. Proceedings*, volume 9152 of *Lecture Notes in Computer Science*, pages 111–120. Springer.

Kolovos, D. S., Paige, R. F., and Polack, F. (2006). The epsilon object language (EOL). In Rensink, A. and Warmer, J., editors, *Model Driven Architecture - Foundations and Applications, 2nd European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer.

Mikosik, M. (2021). FluentIterable. https://guava.dev/releases/21.0/api/docs/com/google/common/collect/FluentIterable.html.

OMG (2014). *Object Constraint Language*. OMG, Needham, MA, formal/2014-02-03 edition.

OMG (2015). *Meta Object Facility (MOF) Version 2.5*. OMG, Needham, MA, formal/2015-06-05 edition.

OMG (2016). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.3*. Needham, MA, formal/2016-06-03 edition.

Samimi-Dehkordi, L., Zamani, B., and Kolahdouz-Rahimi, S. (2018). EVL+strace: a novel bidirectional model transformation approach. *Information and Software Technology*, 100:47–72.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Boston, MA, 2nd edition.

Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., Visser, E., and Wachsmuth, G. (2013). *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.

Völter, M., Stahl, T., Bettin, J., Haase, A., and Helsen, S. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

Westfechtel, B. and Buchmann, T. (2018). Incremental bidirectional transformations: Comparing declarative and procedural approaches using the families to persons benchmark. In Damiani, E., Spanoudakis, G., and Maciaszek, L. A., editors, *Evaluation of Novel Approaches to Software Engineering - 13th International Conference, ENASE 2018, Funchal, Madeira, Portugal, March 23-24, 2018, Revised Selected Papers*, volume 1023 of *Communications in Computer and Information Science*, pages 98–118. Springer.