CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Creating a Bi-Directional Source-to-Source Compiler Using MDE Transformation Techniques

A Proof-of-Concept Using the General-Purpose Programming Languages COBOL and C++

Sebastian Blomberg and Joel Severin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Creating a Bi-Directional Source-to-Source Compiler Using MDE Transformation Techniques

A Proof-of-Concept Using the General-Purpose Programming Languages COBOL and C++

Sebastian Blomberg and Joel Severin

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Creating a Bi-Directional Source-to-Source Compiler Using MDE Transformation Techniques
A Proof-of-Concept Using the General-Purpose Programming Languages COBOL and C++
Sebastian Blomberg and Joel Severin

Supervisor: Regina Hebig, Department of Computer Science and Engineering
Examiner: Jan-Philipp Steghöfer, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2017

Creating a Bi-Directional Source-to-Source Compiler Using MDE Transformation Techniques
A Proof-of-Concept Using the General-Purpose Programming Languages COBOL and C++
Sebastian Blomberg and Joel Severin
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

In this thesis, we discuss how a bidirectional source-to-source compiler, for the declarative general-purpose languages COBOL and C++, can be implemented using Model-Driven Engineering (MDE) tools and practices. The outcome of our work is primarily an approach for implementing said bidirectional compiler using formal grammars, bidirectional transformation languages, and a developed concept model. This approach also illustrates how a programmer's intent can be transferred between languages. In order to evaluate the approach, a prototype was realized using Ecore, Xtext, and Medini QVT. In the process, a library for emulation of COBOL data types in C++ was also developed. Finally, we conclude the success of the developed approach and prototype.

Keywords: source-to-source compiler, transpiler, translator, bidirectional transformation, model-driven engineering, COBOL, data type emulation.

Acknowledgements

We would like to thank our supervisor Regina Hebig for steering us in the right direction every time we lost track or had questions. We would also like to thank everyone else that helped us improve our work, including our supervisor Jan-Philipp Steghöfer, and our opponents Peter Eliasson and Jakob Csörgei Gustavsson. Kindly, thank you.

Joel Severin and Sebastian Blomberg, Gothenburg, June 2017

Contents

1
Introduction

A natural part of the technological evolution is the eventual decrease in popularity, and usage, of old technologies. In the field of Software Engineering, one such technology is the general-purpose programming language COBOL (common business-oriented language). Its replacements include more popular languages, such as Java, C, C++, C#, Ruby, Python, recently even JavaScript. Our work aims to ease the transition between COBOL and a replacement language, by transforming old COBOL code to a new language, and then automatically propagating changes made in the new code back to the old COBOL code base. This advancement allows developers with knowledge of only one of the languages to work together, as development in both languages can co-exist. Hopefully, important system and domain knowledge can be transferred between programmers of different generations in the process.

1.1 Motivation in Relation to Industry

Conducting research about transforming COBOL code into a more modern language may be motivated by current circumstances in the industry, including:

- there has been a vast decrease in education of new COBOL programmers [1],
- existing COBOL programmers are nearing retirement age [2],
- a majority of IT managers raise concerns about a current or future shortage of COBOL-knowledgeable developers [2],
- COBOL is not a popular language anymore [3, 4],
- COBOL is still widely used, especially in the financial sector [5] [1].

There are examples of products which transform COBOL into C [6] and COBOL into Java [7]. However, these tools only support migration of COBOL code, meaning that the transformation is unidirectional, and after the transformation is done code is only written in the new language. As a consequence, the underlying system (often a mainframe) may need to be replaced. This presents a challenge for several reasons. First, it may be costly [2]. Second, it may present uncertainty, as there sometimes seems to be a feeling that the old systems, as quoted by a COBOL application

[1] Datamonitor [5] estimates that there were 200 Billion lines of COBOL code in usage 2009, and 5 Billion new lines being added each year. Additionally, they estimated that 90% of the world's financial transactions were processed in COBOL.

manager, "work pretty well" [2]. Third, when existing COBOL developers retire, knowledge is lost. Finally, there might be serious performance issues when migrating to a new system [8]. By utilizing bidirectional translation of code, a COBOL system can remain intact and continue to be developed in C++, thus mitigating some of the factors described above.

Replacing the COBOL language might, however, not be a solution to all problems associated with legacy COBOL systems. While COBOL presents a language barrier, the systems themselves might have a complex design, and the COBOL-knowledgeable workforce not only knows COBOL, but also the domains of these systems.

1.2 Research Questions

The following research questions will guide the thesis work:

RQ1 Which are the most common COBOL constructs used in practice?

RQ2 Which well-known language would be appropriate to use as target, in a transformation from COBOL?

RQ3 How can a bidirectional transformation be applied in order to transform between COBOL (source) and the well-known language (target), such that:

(a) the resulting code is correct according to the target syntax,

(b) at run-time, both programs behave the same in terms of output, given the same input,

(c) changes in the target code base propagate back to the source code base, such that as much of the COBOL code as possible is left intact, with focus on locality of change,

(d) and such that language constructs in the source language are translated in a manner such that *intent*, to the greatest extent, is preserved in the corresponding target language constructs?

where we define *intent* in Definition 1.2.1. RQ2 and RQ1 are questions which need to be answered in order to 1) determine the focus and limitations of our study, and 2) to design the actual transformations in the main research question RQ3.

Definition 1.2.1. Intent *the implicit encoded reason why a piece of code is expressed in one way, when there are multiple ways to achieve the same goal.*

1.3 Purpose

The purpose of our work is to provide an approach for implementing a bidirectional source-to-source compiler using modern transformation techniques, including (but not limited to) techniques from the field of Model Driven Engineering (MDE). Furthermore, a prototype implementation will be presented as a proof of concept for the provided approach.

1.4 Delimitations and Limitations

One part of the outcome of the work we report on is a research prototype. As such, it is neither complete nor fully tested. However, it serves as a proof-of-concept that can be extended and generalized as the field of bidirectional language translation evolves. The prototype does not take code style into account. Instead, focus is put on the actual content of the code and approaches to translate it. Furthermore, although the approaches presented in this thesis are quite general, we make no claim of generality for programming languages dissimilar to the ones supported in the prototype.

A comment on the usefulness of our work should be made. It could be that learning COBOL is a relatively small task, compared to understanding the complex systems written in COBOL. It could be theorized that most of the COBOL systems still in existence today are the most complex ones, since the more simpler ones have already been replaced by modern technologies. Our work will not solve the issue with complex system knowledge, but it could still be useful for a human wanting to adopt such knowledge.

1.5 Disposition

The remainder of this thesis follows the structure of first presenting background theory, and then related work of direct relevancy to our contribution. After that, our methods of conducting work are presented. Next, our contribution is reported on and discussed in chapters 5 to 8. Finally, a conclusion and some ideas for future research are presented. In the part concerning our contribution, we iteratively report on how we solved several problems in several chapters (this constitutes our results), enabling us to reach our goal of answering all research questions.

2
Theory

This chapter introduces the foundations on which the related work (next chapter) of direct relation to our contribution, and our contribution, are built upon. Since our work covers multiple sub-fields of software engineering, a short summary with motivations follows to orient the reader. First, formal grammars are described, which can be used to parse code in text form and generate a model from it. As a specification, formal grammars also allow text to be generated from the grammar rules it encodes, possibly from an existing model. Second, the notion of a model is formalized, with the introduction of metamodels. The mappings between text and model are essential to our contribution and are further elaborated on in the Related Work chapter. Finally, transformations, especially those moving information between different models (as opposed to text), are described. The important property of a transformation being bidirectional (similar to reversible) is explained in depth.

2.1 Formal Grammars

A formal grammar (henceforth referred to simply as a grammar) contains a set of production rules, linking a list of nonterminal and terminal symbols, to another list of such symbols [9]. The terminal symbols represent the elementary symbols of a language, while the nonterminals contain a list of terminals and other nonterminals. Also, the grammar contains a start rule, imposing some ordering on the application of the production rules. Given a specification of a grammar, a certain language can be parsed by a machine. Certain constraints can be applied to the grammar, restricting which types of languages it accepts.

Chomsky [9] proposed classifying grammars using a hierarchy of constraints, limiting the languages it accepts. The hierarchy orders four different classes of grammars, presented from general at the top, to less general at the bottom, in Figure 2.1. A certain grammar class is able to accept more than all languages all less general grammar classes are able to accept. At the bottom of the hierarchy, regular languages are found, e.g. those parseable by a regular expression. Regular languages allow linking one nonterminal to exactly one termianal, optionally either preceded or followed by a nonterminal. One level up, the context-free languages are found. These allow linking one nonterminal to a list of terminals and nonterminals. At the next level, context-sensitive grammars reside, allowing a context of terminals and nonterminals around both sides of a rule (the context has to be the same on both sides of the rule). Finally, the most general class contains the recursive enumerable grammars, allowing any terminals and nonterminals on both sides of a rule, thus

removing the restriction that the context must be the same, as in context-sensitive grammars.

Backus [10] proposed to standardize specification of production rules linking nonterminals to a list of terminals and nonterminals, allowing context-free grammars to be specified by a textual notation now known as the Backus-Naur Form (BNF). Grammars for general-purpose languages are typically expressed by the de-facto industry standard Extended BNF (EBNF) syntax, now standardized as ISO/IEC 14977 [11]. EBNF is an extension to BNF that adds syntactical sugar which allows easier specification of repetition (e.g. the Kleene star `*`, as also found in regular expressions).

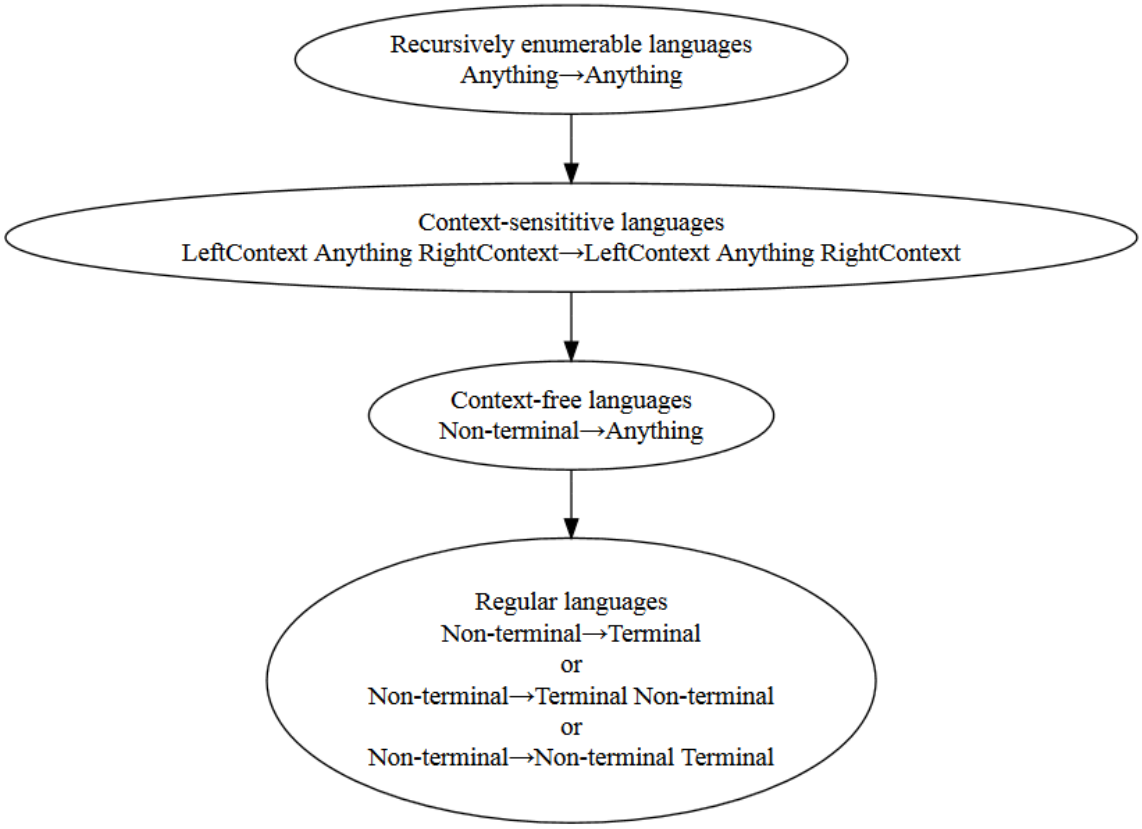

Figure 2.1: Chomsky's hierarchy of languages acceptable by their respective grammar class (replace *languages* with *grammars*). `Anything` is a terminal or non-terminal. `LeftContext` and `RightContext` are terminals or nonterminals, but need to be exactly the same on both sides of the arrow (to define a context).

In order to facilitate parsing of context-free languages by algorithms run on today's computers, Knuth [12] proposed that a suitable subset would be LR grammars, denoting grammars that can parse an input stream by reading it left to right once (without backing up to reconsider previous choices). The algorithms would then exhibit linear complexity with respect to the input length. More specifically, a `LR(k)` grammar is allowed to have `k` tokens of lookahead, meaning that the parser can peek at no more than `k` upcoming lexical tokens in the input stream. LR-parsers have later become known for their other properties: they produce a rightmost derivation,

resolving production rules from right-to-left, and thus produce a bottom-up parse (where the smallest elements of a language are recognized first) [13].

Some additional constraints imposed on the special case `LR(1)`, also known as Canonical LR (CLR), have allowed the less resource-intensive Look-Ahead LR (LALR) and Simple LR (SLR) algorithms to be developed [13]. In fact, these theoretically cover all countable `k`, as it can be proven that for all `LR(k)` grammars, there exists an equivalent `LR(1)` grammar. Additionally, the Generalized LR (GLR) parser builds upon LR parsing, making it less restrictive (but have the same complexity) [14]. These specializations of the LR parser have become popular choices for implementation in compilers, and consequently even language specifications are written with respect to them. For example, the popular parser generator yacc, and its successor Bison, are based on these algorithms [15].

In contrast to LR parsing, a different family of algorithms that have become known as LL parsers has been discovered [13]. These still read the input stream from left to right, but perform a leftmost derivation, resolving rules from left to right, thus producing a top-down parse (where the largest elements of a language are recognized first). For example, the popular parser generator ANTLR uses LL(*)-parsing (from version 4 ALL(*)-parsing), where the star denotes that k may be infinite [16].

For completeness, we mention that parsers are often table-based, meaning that they are built using pre-computed decision tables for which symbol is read, and what is in the look-ahead [13]. There also exist recursive-descent LL-parsers, and recursive-ascent LR-parsers, where the decisions are taken by actual program code, written natively in the programming language together with the parser. Lexers and parsers are typically generated by lexer and parser generators, i.e. tools which, based on a grammar, are able to automatically generate actual parser programs in native source code (C, Java etc.).

Code of a certain programming language is typically read by a compiler using a two-step process of first applying lexical analysis to the input text stream using a regular language, then parsing them with a restricted context-free grammar (e.g. with a LALR parser) [17]. The lexer tokens provide a symbol representation of the underlying text stream, consisting of a name and an optional attribute (for example `(INT, 42)`, when lexing the integer 42). The token names are then interpreted as nonterminals for the next stage, where the parser uses the context-free grammar to produce an abstract syntax tree (AST), for further use by the compiler.

2.2 Metamodeling

The code-centric development approach, i.e. defining programs by writing code, is the prevalent and traditional way of developing software. One explanation for why this is the case might be that the majority of all general-purpose programming languages are defined by grammars, meaning that somewhere along the way code must be written to produce software.

Another approach to describe and develop software is using models. In the field of software engineering, Seidewitz [18] defines a model as "a set of *statements* about some system under study (SUS)", where a *statement* is "some expression about the SUS that can be considered true or false". This definition is rather general and

for a model to be meaningful there needs to exist an interpretation. Seidewitz [18] defines an interpretation as a "mapping of the model's elements to elements of the SUS such that we can determine the truth value of statements in the model from the SUS". For example, suppose we have an Entity-Relationship (ER) model. The model contains expressions of which data to be stored and relationships between collections of these data expressions (entities). If we say that our SUS is a relational database, the interpretation of the model is that an entity is mapped to a table, an attribute to a column, a relationship to a foreign key constraint, and so on. In this case, the statements of the model are true if their mapped counterparts in the SUS exist and conform to the model. If all statements in a model are true, Seidewitz [18] defines the model as being *correct*.

A model can be expressed through the use of a modeling language. For example, in order to express an ER model, there needs to be a definition of how and what it can contain (e.g. entities, attributes, etc.), as well as how the different components relate; this is defined in a modeling language. The Unified Modeling Language (UML), containing language definitions for a wide range of applications, such Class diagrams (models) and State machines, is the predominate modeling language for software.

A modeling language is itself defined by another model, referred to as a *metamodel*. A metamodel expresses statements which any model created in a modeling language needs to adhere to. As such, the interpretation of a metamodel is mappings between its elements to the elements of the modeling language, and therefore a metamodel determines the validity of a model created in a modeling language [18]. An analogy can be made between a metamodel and a grammar. A grammar is a textual specification (a set of rules) of how a piece of code (text) may be written to be valid. Similarly, a metamodel is a specification of how a model, created in a certain modeling language (a set of rules), may be created to be valid.

Since a metamodel is actually a model, it can be expressed in a modeling language, implying that a metamodel can be defined by a meta-metamodel. In turns, that meta-metamodel can be defined by a meta-meta-metamodel. Theoretically, there is no limit of how many metamodels there can exist in a chain of metamodels. However, at some point they are bound to be superfluous, specifying trivialities. In such a case, a metamodel can be used to describe itself.

Within model-driven software engineering, there is a four-layer architecture traditionally used when referring to layers of metamodels, as depicted in Figure 2.2 [19]. At the top layer M3, there is a general (meta) model specified in the Meta-Object Facility (MOF) standard [20]. This M3 model is self-contained, meaning that it can be used to describe itself. An M3 model can also be used to describe an M2 model. The most commonly used M2 model is the UML metamodel, although the MOF standard does not limit the choice to UML. Layer M1 classifies user-defined models, conforming to the metamodel in layer M2. For example, a layer M1 user-defined UML model could be a model for modeling users in a system. Continuing the example, the corresponding layer M0 model would be the instance of the M1 (meta) model, i.e. the data which describes the actual user. See the right-hand side in Figure 2.2 for an illustrative example.

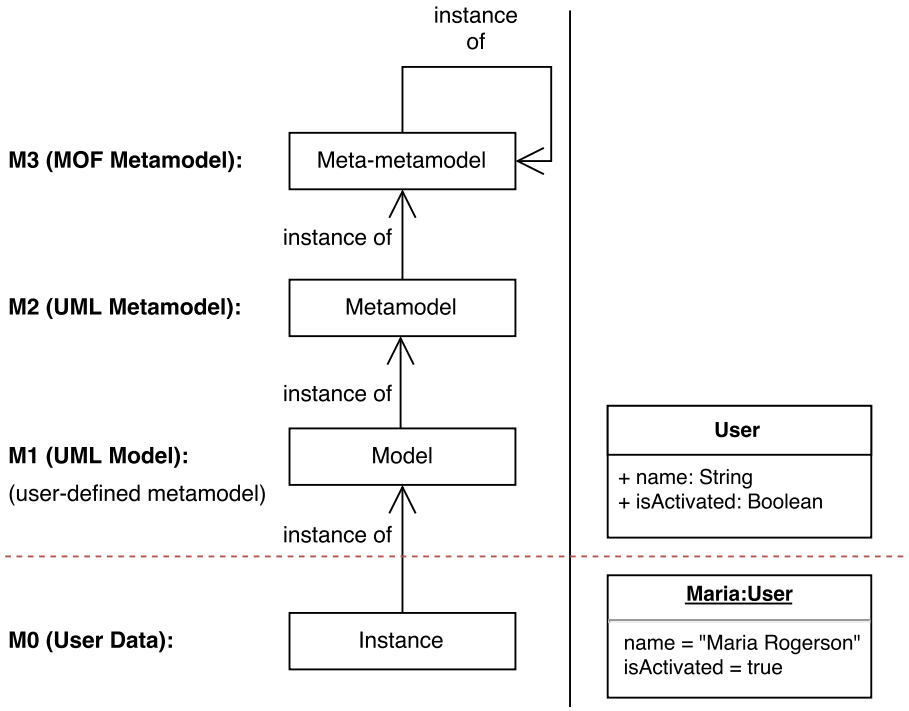

Figure 2.2: An illustration of the *four-layer architecture*. The left-hand side of the figure illustrates the relations between the models of the four layers, whereas the right-hand provides an example of usage.

2.3 Transformations

Transformations is a mechanism for establishing consistency between, or create new, artifacts through the means of a relation, where an artifact might be for example a text, a model, or any other kind of formally defined information. Transformations are used in a variety of areas including Software Engineering, Programming Languages, and Databases. A simplistic definition of a unidirectional transformation is as follows $r : S \to T$, where r is the function which specifies the relation between S, the set of source artifacts, and T, the set of target artifacts [21]. By applying the transformation, a target t that complies with the source s, according to the relation r, can be produced. Consistency between a source and a target is considered true if and only if $t = r(s)$. Note that in grammar-related literature, such as [22], a transformation may be referred to as a translation when it is performed on text. In general, it is intuitive to think of a translation as a transformation, since a translation transforms text, although context-dependent meanings might apply in some cases. We will use translation and transformation interchangeably when referring to the end-to-end process of translating code; note that a translation may consist of several transformations that are not text-based, as long as the outcome is text.

Apart from unidirectional transformations, there are also bidirectional transformations where a relation between two (or more) artifacts applies in both directions. Borrowing from Stevens [21], a bidirectional transformation can be defined by a statement of consistency $R(s, t)$ which is deemed true if the source s and the target

t is consistent in regards to relations R. For each relation in R, there are two directional transformations, one in each direction, such that $\overrightarrow{R} : S \times T \to S$ (forward direction) and $\overleftarrow{R} : S \times T \to T$ (backward direction); that is, given a pair of models s and t, the transformation should be able to restore consistency in any direction according to the relations R. It should be noted that in a truly bidirectional transformation, there is no distinction between which artifact is the source and which is the target. However, in literature, the terms are still used when describing bidirectional transformations.

Bidirectional transformations are inherently more complex than unidirectional transformations. One of the main issues faced with bidirectional transformations is dealing with non-trivial types of relations. A relation between two artifacts in S and T, respectively, is *bijective*, if for every artifact $s \in S$ there exists exactly one unique artifact $t \in T$, i.e. there is a one-to-one mapping between artifacts (see Figure 2.3a). Bijective relations are easy to handle, in the sense that there is always an obvious way to calculate the inverse of a bijective transformation, i.e. the transformation in the other direction. However, a bijective constraint may be too restrictive in many cases [21]. *Surjective* relations, on the other hand, allow for more flexibility. A relation is surjective if for every $s \in S$ there exist an artifact $t \in T$ (not necessarily unique), i.e. there exist many-to-one mappings (see Figure 2.3b). However, there is no definite way of calculating the inverse of a surjective transformation, since the inverse of a many-to-one transformation is a one-to-many, where any of the options are valid. We refer to this as the *general case*, see Figure 2.3c. Henceforth, we refer to relations of a surjective and general case nature as *non-bijective* relations.

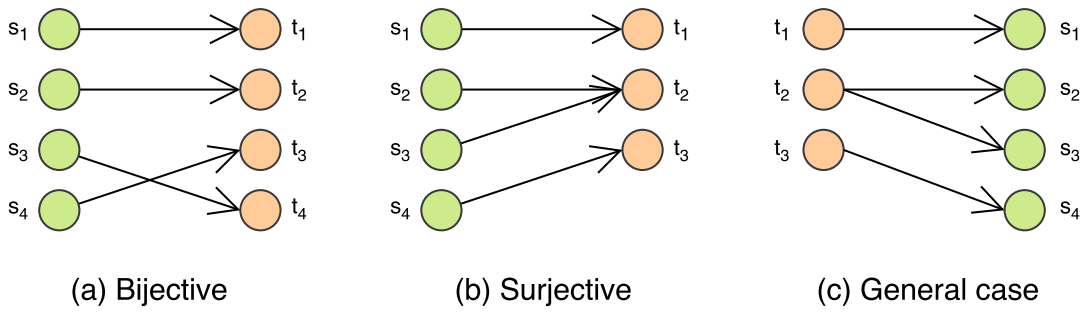

Figure 2.3: An illustration of the types of relation in transformations. (a) shows a bijective relation, which is invertible. (b) shows a surjective relation $S \to T$, which turns in to the general case (c) in the opposite direction $T \to S$.

Although applicable in more places, non-bijective relations present problems when restoring consistency. For example, which of the alternatives in a one-to-many mapping to use; this is a matter of solving *ambiguities*. In some cases, it might not matter since all are valid solutions, but in other cases it does. Consider, for example, if s_2 has been transformed into t_2 in the situation in Figure 2.3b. There are then 2 possible ways to transform t_2 into an artifact part of S, namely either s_2 or s_3, as illustrated in Figure 2.3c. In some cases, e.g. if no changes were made to t_2, it might be important that s_2 is chosen, since s_2 was the original artifact before transformation.

2.4 Bidirectional Transformation Languages

There are several approaches towards implementing a bidirectional transformation. Perhaps the most natural method that comes to mind is to use a pair of unidirectional transformations such that $r_1 : S \to T$ and $r_2 : T \to S$, forming a relation r. The problem with this approach is the absence of consistency validity. For example, consider r_1 to be a valid transformation. There is no guarantee that r_2 is compliant with r_1 and produces the intended result according to the relation r, since it is specified separately from r_1. Poskitt et al. [23] propose a formal way of addressing this issue by automatically translating a pair of unidirectional translations to graph transformations, i.e. states of computations presented as graphs, and nested conditions, in order to apply graph transformation validation techniques to ensure consistency. However, their work is not yet available as a tool for public use.

Another approach for implementing bidirectional transformations is using a transformation language which supports bidirectional transformations. Such languages are able to express bidirectional transformations as one unit, i.e. they are able to express the forward and the backward transformation simultaneously. The benefit of this is that transformation validity is ensured by construct [24]. There is a variety of different bidirectional transformation languages and tools operating on different artifacts (e.g. models, text). Their methods of transformation also vary, which may give substantially different results between implementations. The remainder of this section is used to describe the most common of these transformation methods, as well as a few bidirectional languages which use them.

In order to compare bidirectional transformation languages, we differentiate between two types: those which are able to support model synchronization and those which are only able to generate new models. Supporting model synchronization entails that information from an old version of a model (artifact) can be used to recreate, i.e. synchronize, a new version, meaning that potentially less information is lost.

2.4.1 Triple Graph Grammars

One of the common methods for implementing bidirectional languages is Triple Graph Grammars (TGGs), as introduced by Schürr in 1994 [25]. TGGs use graph grammars, whereby the artifacts to be transformed (source and target) are interpreted as graphs, along with an intermediate correspondence graph which links the source and the target (i.e. defines the relations). Through the use of these graphs, a language implementing TGGs can then generate the forward and backward transformations, as well as check consistency.

Three bidirectional languages, all based on TGGs and operating on EMF-based models (as artifacts), are eMoflon [26], MoTE [27], and TGG Interpreter [28]. Their specification methods are similar and follow a general TGG rule format. Furthermore, they all support model synchronization. However, their implementations differ. MoTE requires all TGG specifications to be conflict-free [29], meaning that all relations are required to be bijective. TGG Interpreter, on the other hand, does not require specifications to be conflict-free, nor does it guarantee that a valid (or

complete) model will be produced, as it does not implement any lookahead or backtracking; as such, if one wrong decision is taken during the transformation, the transformation will continue down the erroneous path. eMoflon, like TGG Interpreter, does not either require specifications to be conflict-free. Unlike TGG Interpreter, however, eMoflon implements local look-ahead, guaranteeing that a valid model, but not which valid model, is produced.

In a comparison of TGG tools, Greenyer [29] concludes that MoTE should be used in situations where conflict-free specifications suffice, due to its efficiency. eMoflon should be used in situations which require look-ahead/backtracking, but then comes with an overhead that affects performance. The benefit of TGG Interpreter is preservation of information during transformations.

2.4.2 QVT Relational

Unlike Triple Graph Grammars, QVT Relational (QVT-r) is not a method for implementing bidirectional transformation, but rather a language specification dictating how transformations should be specified and the intended outcome of transformations; how the actual transformations are implemented is not exactly specified. QVT-r is part of the MOF-based QVT (Query View Transformation) specification [30], published by the Object Management Group. QVT-r takes a similar high-level view of defining transformations as TGGs and Stevens [31] notes that the QVT core language definition is clearly influenced by TGGs.

According to Stevens [31], the QVT-r standard is somewhat ambivalent towards allowing non-bijective transformations. She argues that the standard can be misinterpreted and does not make explicit that non-bijective transformations are allowed. However, she does show that non-bijective transformations are possible to implement, according to the standard, although the expressiveness of those has its limitations.

There are several tools, with different underlying implementations, that implement the QVT-r specification. Medini QVT [32] is perhaps the most mature one. Operating on EMF-based models, Medini QVT can be run as standalone Java program or as an Eclipse plugin (with debug support). Medini QVT does not support model synchronization, meaning that a new target is generated from scratch during every transformation. As such, transformations in Medini QVT can be denoted as $\overrightarrow{R} : S \to T$ and $\overleftarrow{R} : T \to S$, rather than the more general definition, presented in Section 2.3, which has both domains S and T as input parameters

Another tool, based on EMF-models, which implements the QVT-r specification, is Echo [33]. Echo implements transformations by translating models and relations into Alloy [34], which contains a model finder and satisfiability (SAT) problem solver. A SAT problem is set up for Alloy to solve with parameters to ensure the principle of least change, meaning that a transformed model will be as close to its original as possible. Therefore, Echo supports model synchronization through its least-change approach. Notable is also that Echo is able to produce several solutions, each further away from the least-possible change. Furthermore, Echo also takes OCL (Object Constraint Language) constraints into account when finding appropriate models, meaning that the outcome will always be valid with regards to

its metamodel. In comparison, Medini QVT does not take OCL constraints into account, meaning that it may produce invalid models. However, using a SAT solver naturally results in a larger performance overhead.

The Eclipse Foundation is working on its own implementation of QVT-r, included in Eclipse QVTd[35]. While a rudimentary implementation is already released, the full version is planned to be released in July, 2017.

Another tool which does not implement the QVT-r standard, but has a most similar syntax to that of QVT-R and works in a similar manner, is Janus Transformation Language (JTL) [36]. JTL works on EMF-based models and addresses change propagation through means of Answer Set Programming; when a change is detected in the source, a constraint (SAT) solver is used to find the ideal solution for a consistent target. If there is more than one solution, JTL is able to return all. Principally, JTL is comparable to Echo, although their underlying implementations differ.

2.4.3 Text-based approaches

The approaches presented so far have all been based on models, in the sense that they operate on models defined by a metamodel. There are also a set of bidirectional transformation tools which operate on text-based artifacts, e.g. XML (Extensible Markup Language) documents. As noted by Stevens [21], it might seem that model-based transformations should be capable of handling XML documents, since models are defined in XMI (XML Metadata Interchange) files and both formats share many properties. However, there are differences (e.g. handling of identifiers), which make these approaches incompatible.

biXid [37] is an XML-to-XML bidirectional transformation language, based on a programming-by-relation approach of specifying transformations (similar to that of QVT-R). biXid supports non-bijective relations but will choose a solution non-deterministically in cases where ambiguity exists.

XSugar [38] is another text-based bidirectional transformation language, able to transform between XML and a self-defined text format. XSugar requires a context-free grammar to be specified for the text format to transform to. Given such a grammar, XSugar is able to map elements of an XML specification to those of the grammar, thereby bidirectionally transforming text between the two formats. The limitation of this approach is a lack of support for non-bijective relations.

Another text-based bidirectional transformation language is Boomerang [39]. In comparison to XSugar, Boomerang is generally applicable to self-defined text formats and includes a rich set of features for writing transformations. Transformations in Boomerang are written as *lenses*. A lens is a bidirectional program operating between two domains S and T. A lens requires that T is an abstraction of S, meaning that all information in T must have corresponding representations in S (but not the other way around). A lens mapping between the two domains consist of the following operations [39]:

- $Get : S \to T$
- $Put : T \to S \to S$

- $Create : T \rightarrow S$

which have to obey the following laws:

1. $Put\ (Get\ s)\ s = s$, meaning that any information lost when transforming an artifact s to an abstract artifact t, must be restored when transforming back,
2. $Get\ (Put\ t\ s) = t$ and
3. $Get\ (Create\ t) = t$, both meaning that there should be no loss of information when transforming from the T domain to the S domain and back again.

As such, all transformations obeying the laws of lenses are well-behaved, although this comes at the cost that the T domain must be an abstraction of S. In Boomerang, a lens can consist of a composition of other lenses. Boomerang provides a basic set of lenses for transforming strings. Based on those, user-defined lenses can be written in an approach that can be described as a mix of specifying grammar and relations. Furthermore, Boomerang supports non-bijective relations in the sense that when there is ambiguity (which only happens when transforming from $T \rightarrow S$), the programmer can control how consistency should be restored.

3
Related Work

This chapter introduces some research of direct relevancy to our contribution, building upon the foundations laid out in the previous Theory chapter. Two important research areas are summarized: how the technological spaces of grammars and metamodels can be bridged, and how bidirectional transformations between general-purpose languages can be performed.

3.1 Bridging Grammars and Metamodels

Grammars and Metamodels are two different methods of expressing structure for information (code and models, respectively), and they belong to and are treated in two different technological spaces [40]: *grammarware* and *modelware*. Grammars tend to be used for most general-purpose programming languages. In particular, context-free grammars (e.g. BNF), along with plain English for inexpressable semantics, are often used to specify programming language definitions. Equivalently, metamodels are used to describe the semantics of models. While a programming language may theoretically be specified in models, as is the case with some Domain-Specific Languages (DSL), it is not commonly the case for general purpose programming languages. To fully utilize strengths of both grammarware and modelware, it would be beneficial for information specified in one technological space to be available in the other.

Recalling Seidewitz's [18] definition of a model, it may seem trivial that a model can represent code, i.e. the SUS is actual programming code and the interpretation is that if there exist a language construct in the model, it will also exist correctly (in various different aspects) in the code. In a similar manner, a metamodel can be thought of to represent a grammar. While this might seem plausible in theory, the realization of such a bridge between the two technological spaces presents a handful of issues.

Alanen and Porres [41] studied the relationship between context-free grammars and MOF metamodels. In their work, they defined relations between BNF and MOF-defined metamodels in both directions. The basic relationship is one where a rule in BNF is converted to a class in the metamodel. They found that an arbitrary BNF grammar can be converted to a metamodel using an algorithm operating on the M2 layer; the resulting metamodel may not be very intelligible, depending on the grammar. In the opposite direction, however, they found that the set of metamodels that can be converted to BNF grammars is restricted. Metamodels intrinsically contain more information than BNF grammars, which presents the

problem of maintaining that information when transforming back to a metamodel from a BNF grammar.

Inspired by the work of Alanen and Porres, Wimmer and Kramler [42] propose a theoretical framework for bridging the two technological spaces. Their work extends that of Alanen and Porres by defining relations in the M1 layer, meaning that programs and models can be transformed. Furthermore, Wimmer and Kramler tries to enhance generated metamodels by a series of transformation rules which eliminates redundancy. They also introduce the concept of a change model which can contain annotations that dictate properties of a metamodel derived from the grammar. These annotations include specifying which attributes should be treated as references instead of compositions, what identifiers should be used for referencing, and data type specification for attributes (e.g. string, int, boolean). By applying the change model, the generated metamodel will be richer in details. Kunert [43] presents a similar framework to that of Wimmer and Kramler, although instead of having a change model, Kunert extends EBNF to support specification of annotations directly in the grammar. For example, terminals which should not be included in the abstract syntax tree, and therefore neither in a metamodel generated from the grammar, can be marked to be excluded from the generated metamodel. Although the problem of maintaining information when converting from a metamodel to a BNF-based grammar and back still remains using annotations, the set of supported metamodels becomes less restrictive, since the grammar contains more shared information with MOF metamodels.

3.1.1 Xtext

While the frameworks by Wimmer and Kramler [42] as well as Kunert [43] are largely experimental, there is a more mature framework included in the Eclipse project, called Xtext, which is able to bridge the grammarware and modelware technological spaces. Xtext is a framework intended for creating textual DSLs [44]. It shares similar properties to Kunert's framework, namely the ability to annotate grammars in order to describe the generated metamodel. One difference to Kunert's framework is that Xtext takes the approach of specifying what should be included, rather than what should be excluded. That is, Xtext's grammar syntax, which has a similar style to that of EBNF, is quite rich in details about what should be included and how entities should be represented (e.g. reference/containment, collection or singular objects, data types, names, etc.) in the generated metamodel. The example in Figure 3.1 shows an example of Xtext's grammar syntax. Xtext is also able to generate a grammar based on an already existing metamodel, essentially taking the metamodel and treating it as a metamodel for the abstract syntax tree. Furthermore, Xtext includes support for model-to-text transformations in both directions, i.e. serialization and deserialization of a model, and is able to generate a textual editor for a language described by a Xtext grammar.

Usage	EBNF	Xtext
Definition	=	:
Concatenation	,	Whitespace
Termination	;	;
Alternation	|	|
Optional	[...]	(...)?
Zero or more	{...}*	(...)*
One or more	{...}+	(...)+
Grouping	(...)	(...)
Exception	-	!
Terminal String	"..."	'...' or "..."
Enumeration	name = "opt1" | "opt2"	enum name: value1="opt1" | value2="opt2"
Simple assignment	A = B C	A: bAttr=B cAttr=B
Collection assignment	A = B+	A: bs+=B+

Table 3.1: A summary of the relationships between notations in EBNF and Xtext grammars, as found by Yue[47]

```
ThisIsARule returns AClassType :
    key=IDENTIFIER  // an attribute named 'key', matches an IDENTIFIER (lexer token)
    (alternativeKeys+=IDENTIFIER)+ // a list of alternative keys (at least one)
;
```

Figure 3.1: An illustration of basic Xtext syntax. The figure shows a rule of type `AClassType`, with one attribute `key` and a collection `alternativeKeys`.

Although Xtext was initially intended for specifying DLSs [44], there are no theoretical limitations for using it to specify existing general-purpose programming languages. The limitations are rather those of the underlying parser generator. Xtext uses ANTLR 3 [45] in order to generate a parser which parses textual input (i.e. code). ANTLR 3 generates an LL(*) parser [46], as explained in Section 2.1. One drawback of using LL(*) is the non-tolerance of left-recursive rules - a problem which needs to be addressed as quite often existing general-purpose programming languages are defined in BNF-based grammars, which allow left recursion. Another problem is addressing the lack of meta information available in the existing BNF-related grammars. Xtext, for example, requires names for all attributes, as well as specification of cardinality (i.e. singular or list), as examplified by Figure 3.1.

Yue outlines the relationship between notations in EBNF and Xtext in [47]. Table 3.1 summarizes some of these relationships. Additionally, Yue [47] presents a set of strategies for dealing with patterns allowed in EBNF, such as left recursion and common factors (e.g. A = B | B C, where B is the common factor), but not in Xtext due to the underlying parser generator.

Yue [47] does not suggest an automatic process for converting a EBNF gram-

mar to Xtext format; nor does she deal with the lack of information (e.g. references, data types) which is required to generate a fully fledged Xtext grammar from the Xtext grammar. While there does not exist a fully automated solution to convert a complete Xtext grammar today, there exists semi-automated ones. Bergamyr and Wimmel [48] present such an approach, where they define an Xtext grammar for EBNF itself, allowing an arbitrary EBNF grammar to be converted into a model. They then apply model-based transformation techniques in order to translate an EBNF model into an Xtext model (grammar), based on similar relationships as presented in Table 3.1. It is unclear if they automatically refactor left recursive rules, but their article mentions dealing with difficulties related to LL(*)-based parsing, of which existence of left-recursive rules is a primary problem. Although they are able to transform many types of entities automatically, cross references still have to be handled manually. They deal with this in a similar manner as Kunert [43], by extending their EBNF grammar to allow annotating attributes as references. However, they deal with refinement of types for cross references in the Xtext grammar automatically by example, meaning that if given sample code containing cross references, a mechanism is able to figure more specific types for the cross references in the grammar.

All methods of bridging grammars and metamodels mentioned above will most likely require additional refinement to produce a useful metamodel. The reason for this is that most grammars are not intended to be used as metamodels and therefore they contain less information than required to produce a useful metamodel. Therefore, going from the grammarware technological space to the modelware technological space may entail a certain overhead in terms of refining a generated metamodel

3.2 Translation of General-Purpose Languages

The idea of translating between general-purpose computer languages arose during the 1950s-1960s, when code was highly dependent on the computer type it operated on, and therefore not very portable [49]. Early comprehensive attempts to solve this problem involved the construction of an intermediate universal language, UNCOL [50]; the idea being that every programming language had a translator which translated to UNCOL, and every computer had a translator which translated UNCOL into machine code. However, this idea was never fully realized [49]; although the later-created LLVM is built on a similar idea [51]. In the remainder of this section, subsequent approaches to primarily bidirectional language transformation are presented.

3.2.1 Early Attempts at Bidirectional Translation

Early attempts at *bidirectional* source-to-source translation include the work of Albrecht, Krieg-Brückner [52], et al. in which they translate between Ada and Pascal. Their work translation consists of a chain of sub-translation in the following manner (illustrated in Figure 3.2): Ada is translated to a subset of the Ada language, termed AdaP. AdaP is translated to an intermediate representation. The intermediate representation is translated to a subset of the Pascal language, termed PascalA (since

PascalA is a sublanguage of Pascal, no transformation from PascalA to Pascal is required). The opposite order applies for translation in the other direction.

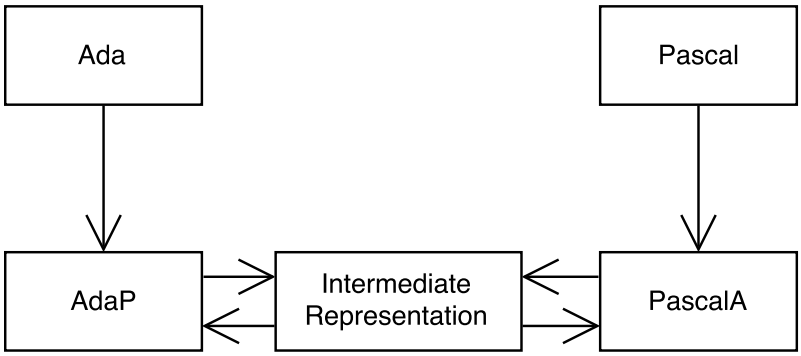

Figure 3.2: An illustration of Albrecht et al.'s [52] approach to bidirectionally translate between Ada and Pascal Code. The arrows each represent a transformation. Note that AdaP $\subset$ Ada and PascalA $\subset$ Pascal.

The sub-languages AdaP and PascalP are defined by Albrecht et al. [52] to only include language constructs which have bijective relations to the corresponding language constructs of the other sub-language, i.e. such that there only exists one one-to-one mapping between a language construct and its respective counterpart. For example, assignment of data to a variable is possible in both languages and is done in a similar manner which, if translated, could have a bijective relation. Therefore, variable assignment is included in the sub-languages AdaP and PascalP. However, array assignment is only possible in Ada, and not in Pascal. Therefore, array assignment is not included in the sublanguages. An array assignment can, however, be expressed as multiple variable assignments each accessing one element of the array. As such, array assignments in Ada can be translated by first decomposing an array assignment in Ada into multiple variable assignments in AdaP. AdaP is then translated in the rest of the translation chain. The purpose of the transformations from Ada and Pascal into their respective sublanguages is therefore to convert language constructs which have no direct one-to-one mapping to the other language into those which have.

In order to implement the bidirectional transformation between the two sublanguages, Albrecht et al. [52] use an intermediate representation to simplify the transformation. This intermediate representation is the unifying abstract syntax tree (AST) of both sublanguages, and contain only the necessary information to express programs in both sublanguages, i.e. unnecessary syntactic and logical information is removed. In total, three different transformations for each language (six in total) are used, as depicted in Figure 3.2. It is worth noting that there are no transformations from a sublanguage to its full version, meaning that once a program has been translated, advanced language constructs will have been replaced. It should also be noted that Ada and Pascal are not fully compatible and therefore Albrecht et al. [52] do not support the full language specification to be translated.

3.2.2 The Idea of Concepts Common to Code between Languages

Krieg-Brückner, one of the authors of [52], later generalized the ideas of the Ada-to-Pascal translation method in [53]. Krieg-Brückner proposes that *concepts* that are more general than specific language constructs should be identified when translating code, first to translate to a subset of the source language (compatible with a greatest-common-divisor intermediary), and then into the target language. As an example, an advanced Fortran IF-statement is translated into a basic variant using IF and GOTO, still in Fortran. The simple IF-and-GOTO construction is then translated into equivalent code in Pascal. Thus, the advanced IF-statement, that was supported only by Fortran, could be emulated by simpler code that had a direct counterpart in Pascal. Krieg-Brückner also proposes that certain concepts might be emulated in the target language (for example, exceptions in Ada have no direct counterpart in Pascal). A discussion follows about which abstraction level to choose: should all higher-level constructs be mapped to IF-and-GOTO constructions, and all types be mapped to a chunk of raw memory (e.g. a byte array)? Or, should the higher level concepts be kept as far as possible in the translation architecture? We will treat this questions in our result section, with some support by Wing. Wing [54] introduced the idea of *computational thinking*, describing how humans can be trained into thinking like a computer, abstracting problems with models and then algorithmically solving them.

3.2.3 Formalizing Bidirectional Translation

Building on the work of Albrecht et al. [52], Yellin [55] developed a method for translating general-purpose programming languages, based on invertible attribute grammars (AGs) and an intermediate language. AGs, a concept proposed by Knuth [56], is a formal way of expressing the semantics of a context-free language, by associating attributes with production rules of a context-free grammar. The value of these attributes can be set by assignment of other attributes or semantic functions, and when evaluated, an attribute grammar can, for example, be used in the translation of code.

Yellin and Mueckstein [57] proposed a method for automatically inverting attribute grammars that specify translations between two languages. Using this method, a translation (transformation) can be written from one language to another and the translation in the other direction will be automatically generated, with validity ensured by construct. As such, this method shares many similarities with simpler bidirectional transformation languages (and perhaps it could even be considered as one). For an attribute grammar to be invertible, it has to be specified on what Yellin and Mueckstein [57] refer to as *restricted inverse form*, which limits its expressiveness (see [57] for more details). However, this does not seem to have a high impact when translating most general-purpose programming constructs.

With invertible attribute grammars as the basis for translation between languages, Yellin [55] proposed a general architecture for source-to-source translation between two or more programming languages, as illustrated in Figure 3.3. Every

language construct in language A is translated to one language construct in the intermediate language I. Likewise, every language construct in language B is translated to one language construct in I. Every language construct in I may be translated into one of *several* language construct in A and B respectively. One benefit of having this intermediate language is that adding a new language to the translation process requires only one additional translation to be written, namely, one from the new language to the intermediate language (e.g. T_C in Figure 3.3), since the inverse is automatically generated (e.g. T_C^{-1}). If translating between n languages, only n transformation would need to be written in total, compared to $\binom{n}{2}$ (one for every pair of languages) had an intermediate language not been used.

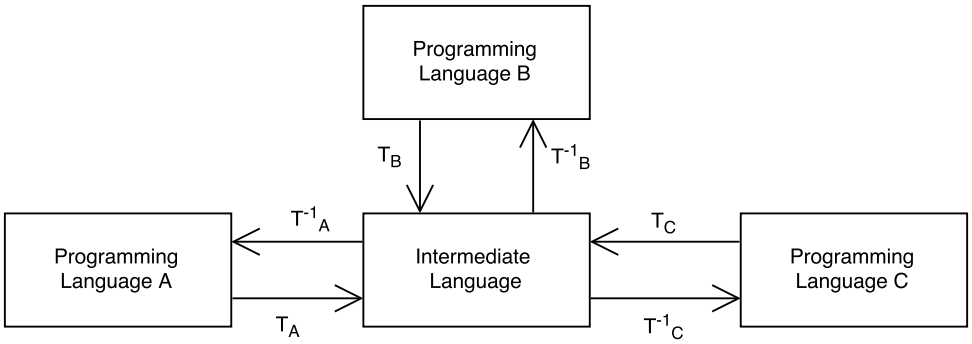

Figure 3.3: An illustration of Yellin's [55] architecture for bidirectionally translate between multiple languages. This figure is based on the work of Yellin [55].

The intermediate language I is defined by Yellin as the *canonical form*, meaning: "1. Every program (in each source language) must be expressible as a program in the canonical form, and 2. every canonical form program that is the image of one translation must be expressible as the image of every other translation." [55]. Yellin then goes on to say that a preferable strategy in order to pick a canonical form is to select the *greatest common divisor* of all languages involved in the translation, meaning that every language construct that has corresponding construct(s) in all other source languages should be included. This way, as much as possible of the program structure as possible is kept intact during a translation. Otherwise, since virtually all general-purpose languages are Turing complete, many language constructs could be written in the form of another (e.g. a loop could be written with GOTO statements and thus only GOTO would be needed in the intermediate language), but this could possibly result in unreadable code after translation.

One benefit of using an intermediate language selected by *greatest common divisor* is that the transformation becomes more concept-centric, as described in Section 3.2.2. Compared to the approach taken by Albrecht et al. [52] consisting of two layers of translations, every language construct which does not exist in both languages can be mapped directly to a viable representation in the intermediate language. For example, in C, a numeric variable can be incremented in different ways, e.g. (i) `i++`, (ii) `i += 1`, (ii) `i = i + 1`. However, when translating to a language which only permits expression assignments (iii), the greatest common divisor is (iii). This scenario gives rise to a many-to-one mapping, and a one-to-one mapping, as illustrated in Figure 3.4. Albrecht et al. [52] would deal with this situation by first

transforming alternatives (i) and (ii) to (iii) in a sublanguage of C. Their sublanguage representation is then translated to the other language. As such, Yellin's [55] approach requires less transformations. It does, however, require that any many-to-one mapping is managed when translating in the other direction. For example, which alternative (i), (ii), or (iii) should be chosen when transforming from the other language back to C? Yellin solves this by statically selecting one alternative as the only alternative and marking the other production rules in the attribute grammar as non-invertible (illustrated by the direction of the transformation of alternative (i) and (ii) in Figure 3.4).

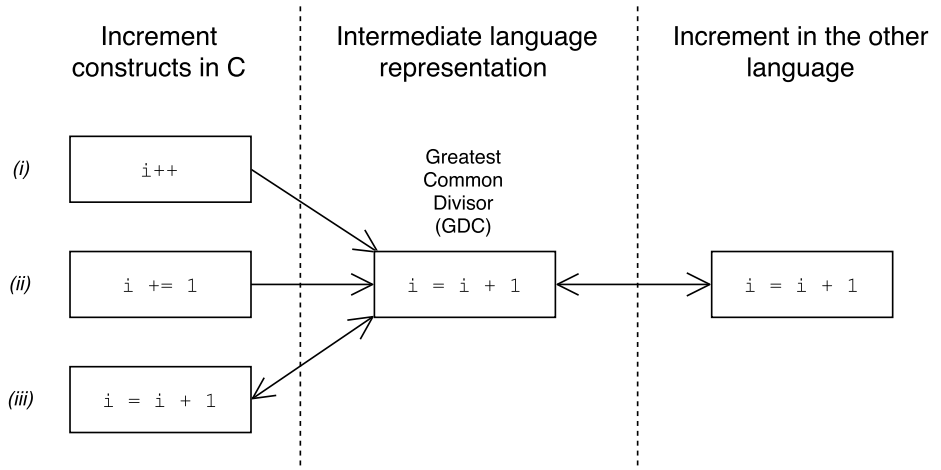

Figure 3.4: An example illustrating a translation scenario with multiple ways to express an increment statement in one language (left), and only one in another language (right). The GCD of these languages is `i = i + 1` (middle).

Yellin [55] addresses an issue with defining the intermediate language based on greatest common divisor, namely that if another source language with a lower level of abstraction is added, the entire intermediate language might need to be redefined. As such, he discusses another form, *least common multiple*, which entails that all high-level constructs found in any of all source languages are included in the intermediate language. The burden, instead, is then to write translations which can describe every high-level construct not found in a language, based on the low-level constructs that exist. Furthermore, the intermediate language may become incomprehensible and hard to grasp.

4
Method

As a general discipline, design research was used as a research methodology due to the specific circumstances of our work; specifically solving the real-world problems identified in section 1.1 based on theory and real-world context, and refining existing design disciplines. In order to answer all research questions, results were reached by iteratively solving smaller problems and combining them. The result chapters outline the main problems explored:

- how COBOL is used in practice,
- which popular language is suitable for translation together with COBOL,
- how such a translation can be performed bidirectionally,
- and what strategies are needed for parts not directly translatable.

Systematically, we followed the guidelines on design research defined by Hevner et. al. [58] (refer to Table 1 in their paper):

Guideline 1: Design as an Artifact Our work produced a viable artifact in the form of a working source-to-source compiler prototype.

Guideline 2: Problem Relevance Our work is of relevancy to the industry, as motivated in the Introduction chapter.

Guideline 3: Design Evaluation The artifact was evaluated, as described later in this chapter, on its *utility, quality, and efficacy.*

Guideline 4: Research Contributions We present a research contribution in the area of bidirectional transformations on general-purpose programming languages, using higher-level concepts (as presented as results in Chapter 8).

Guideline 5: Research Rigor The results were reached and evaluated by scientifically applying the methods described later in this chapter.

Guideline 6: Design as a Search Process We extensively used and combined available artifacts, in the form of already developed software tools. Where no suitable software artifact was available in order to solve a problem, we developed one.

Guideline 7: Communication of Research This thesis serves to introduce and develop the field. While is is mostly technically oriented, there are some parts of relevancy to management. Specifically, the importance and implications to the industry have already been described in the Introduction.

4.1 Survey of Cobol Language Construct Frequency

In order to survey how COBOL is used in practice, a tool was developed that can count occurrences of the different language constructs available to COBOL programmers in actual program code. The formal grammar that the open-source compiler GnuCOBOL [6] (formerly OpenCOBOL) uses when parsing programs for compilation, was used as the baseline for what language constructs the tool can recognize. For analysis, the full source code of the open-source program Applewood Computing Accounting System [59] was used. The analysis results ranked usage of language constructs, by occurrence, in the analyzed code base. The results are presented in chapter 5.

4.2 Choosing a Target General-Purpose Language

The task of choosing a language to transform COBOL between is many-faceted. First, a requirement has been that the language is popular and general-purpose in nature. Next, different languages exhibit different properties, making them compatible to different degrees. This section describes how we chose the best candidate for our requirements. The results are presented in chapter 6.

First, a list of popular general-purpose languages was compiled, after which a comparison of these languages and COBOL was performed. In the comparison, we chose different criteria to classify the languages. Languages were eliminated from the comparison early on when it was clear they were not suitable for use together with COBOL. These criteria were chosen, according to our best judgment of relevancy, to be the following:

- Static or dynamic typing
- Memory management (allocation on stack, heap, and garbage collection)
- Primitive types
- Classes and objects
- Functions
- Basic syntax (arithmetics, conditionals, branching, loops)
- Pre-processing and meta-programming (macros, templates, overloads)

4.3 Emulating COBOL Data Types in C++

As is perhaps already clear from the heading, C++ was ultimately chosen as an optimal target language. It was also established (in the results, see chapter 7) that some data types cannot be translated directly from COBOL to C++, as there is no direct counterpart. These types can, however, be emulated in code by user-defined data types. A proof-of-concept standard library, providing emulation for all COBOL data types not already supported by C++, was developed with the following goals:

- Bit-perfect storage of data (mirroring the COBOL format).
- Integer formats:
  - Plus operator overloading (showing arithmetic support).
  - Casting operator overloading between the emulated types and native types.
  - Arithmetic overflow callbacks.
- Strings:
  - Proper formatting of string data types, as COBOL formats them (arbitrary COBOL PICTUREs).
- Compound data definition:
  - COBOL RECORD emulation.
  - COBOL REDEFINES RECORD emulation.

Each method/overload on the developed types was tested when implemented, showing that the developed features work. The corresponding result chapter is chapter 7.

4.4 Creating a Model-Driven Source-to-Source Compiler

The general method used for translation in the developed source-to-source compiler can be described in 4 steps: a text-to-model transformation from the source language, a model-to-model transformations to an intermediate model, a model-to-model transformation from the intermediate model, and a model-to-text transformation to the target language. This process is illustrated as an activity diagram in Figure 4.1.

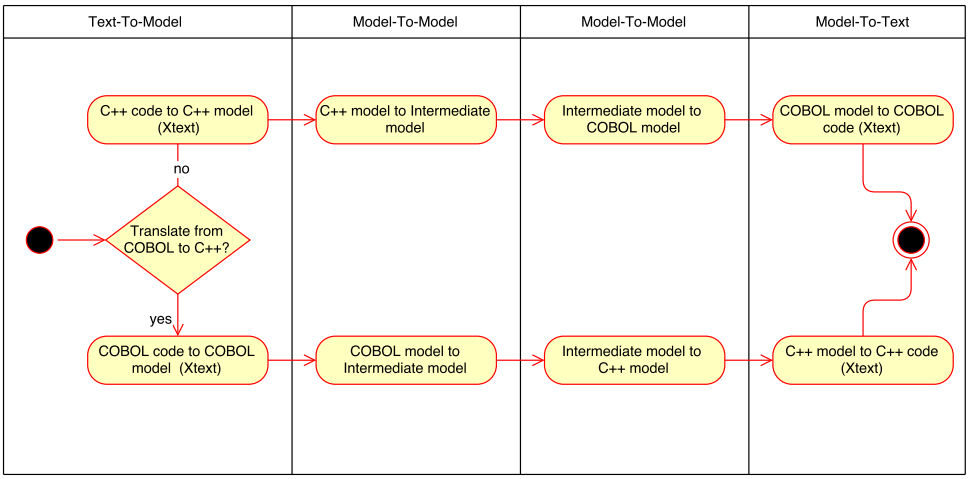

Figure 4.1: An activity diagram showing the transformation flow, including each step involved in the translation process.

In order to implement the model-to-model transformations, the bidirectional transformation language QVT-R was used. The choice of using a bidirectional transformation language, compared to specifying pairs of unidirectional transformations, is motivated by gaining transformation validity by construct. This is a means to partly answer RQ3a and RQ3b. The motivation for choosing QVT-R over the other options (presented in Section 2.4) is threefold:

1. QVT-R is an established language standard, meaning that code written could be run in several tools (interpreters), even eventual future tools which may become more mature over time. In contrast, tools implementing TGGs might have slightly different syntax and setup, and all text-based transformation languages have different syntax.

2. There are tools available that implement model synchronization and handle non-bijective relations. These tools are almost exclusively built on top of Ecore, meaning that any models used will be compatible with any other Ecore-based tools (for example, analysis and diagram tools to improve quality and understating). In short, many opportunities in the MDE field will become available.

3. QVT-R provides the option to specify OCL expressions for handling complex relations and conditions. To our understanding, text-based languages (such as Boomerang) and TGG-based tools are not as flexible in this aspect.

Furthermore, a reason for not selecting a text-based transformation language is uncertainties about whether a C++ and COBOL grammar can be fully specified in such a language.

The remainder of this section presents the steps taken when implementing the prototype source-to-source compiler. Work was guided by small iterations where each of these steps was to some extent included. The outcome of each iteration determined decisions and scope of the next.

4.4.1 Creation of an Xtext COBOL Grammar

Since QVT-R is a model-based transformation language, COBOL and C++ code must be transformed to models in order to apply transformations. To bridge the grammarware and modelware technological spaces, Xtext was used. That is, an Xtext grammar for COBOL was defined and Xtext was used to generate the corresponding metamodel, enabling text-to-model transformations for COBOL code.

The focus of our work is primarily on translation. As such, we chose to reuse and adopt an already existing grammar for COBOL. The grammar chosen [60] was created by Lämmel and Verhoef in [61] and reflects the IBM's VS COBOL II Reference Summary. The grammar is specified on EBNF format and has been tested on over 2 million lines of COBOL code [61].

Lämmel and Verhoef's [61] COBOL grammar was converted to an Xtext grammar using Yue's [47] approaches and rules for conversion between EBNF and Xtext (See Table 3.1). In addition to these rules, another rule one was required: the symbol '||' in Lämmel and Verhoef's grammar denotes any permutation of two rules, which is represented as an unordered group, i.e. the '&' symbol, in an Xtext grammar.

4.4.2 Creation of an Intermediate Model

Like Yellin [55], we used an intermediate representation (model) to implement transformations between the two source languages. This intermediate model was defined to be the canonical form of the two languages. Unlike Yellin, however, we present our own approach, based on common *concepts* between programming languages, for defining the canonical form, in order to answer RQ3d. This approach, together with the resulting model, is presented as a result in Section 8.3.

4.4.3 Specifying Transformations in QVT-R using Echo

The QVT-R transformations between each of the source models were originally intended to be run by Echo. The motivation for this was threefold:

1. Echo provides support for model synchronization through its least-change principle. In practice, this would mean that any changes made to the target model would propagate back and result in a minimally changed source model. As such, RQ3c would, in theory, be answered, since the least-change principle would guarantee only local changes.

2. Echo's least-change principle would guarantee that that any non-transformable information excluded in the transformation would remain intact after a back-and-forth transformation.

3. Echo provides support for generating all possible solutions when synchronizing models, meaning that a user or an automated mechanism could pick the most appropriate solution.

Using Echo v0.3.1, Eclipse Modeling Tools v4.6.2, Eclipse QVTd v0.11, and Eclipse OCL v4.2, we wrote the initial transformations in QVT-R. However, we encountered numerous problems with Echo. First, we discovered that Echo does not support certain Ecore data types present in our metamodels generated by Xtext. Second, when running transformations, several scenarios produced Java exceptions, which required us to debug Echo's source code, and in some cases modify it. Finally, we could not make Echo obey containment rules in metamodels, i.e. Echo treated a containment (composition) as a regular reference. As a consequence, the solutions produced by Echo were invalid. The solutions produced could become massive; that is, Echo could produce solutions which contained classes that were not connected to anything else in the model. As such, when the model grew in size, the SAT problems solved by Echo grew even more in size (since they included all invalid solutions) and took too long to solve. As an example, synchronizing two simple models with 5 classes each, where each class had a small set of attributes and relations to other classes in the same model, could produce a SAT problem with over 2 million variables, taking over 30 minutes to solve. Therefore, we made the decision to abandon Echo.

4.4.4 Specifying Transformations in Medini QVT

Due to the problems faced with Echo, we used Medini QVT to run QVT-R transformations instead. The transformations already written could directly be reused. Due to limited developer literature for Medini and QVT-R in general, we faced several scenarios which were non-trivial to solve. We devised approaches for these scenarios, and together with the resulting transformation specifications, they are described as results in Section 8.4.

A drawback of using Medini QVT is that it does not support model synchronization, meaning that RQ3c will not be fulfilled when running the transformations in Medini.

4.4.5 Creation of an Xtext C++ Grammar

Similarly to the approach of defining an Xtext COBOL grammar, an Xtext grammar for C++ was defined. Lämmel and Verhoef's [61]'s COBOL grammar does not support object-oriented COBOL. As such, we limited our source-to-source compiler to only support the sequential part of C++ (i.e. C-like part), with the exception from a few classes supplied by us to emulate COBOL data types. We found a C grammar defined by Terence Parr [62], implemented in ANTLR3, which was converted to an Xtext grammar. Unlike the COBOL grammar, this C grammar has not been rigorously tested. However, it sufficed to represent the sequential part of C++ which we required. Due to the fact that this C grammar was implemented in ANTLR3, it was trivial to convert to Xtext (since Xtext is built on ANTLR3). Therefore, using this C grammar was deemed preferable to converting a C++ EBNF grammar to Xtext.

4.4.6 Evaluating Results

Due to the limited size of our prototype, we were able to perform manual tests on different combinations of each language constructs to evaluate our approach. Representative examples, based on code from the Applewood Computing Accounting System, were devised and used for evaluating mainly 3 different aspects:

1. Correctness - whether the translation is valid, in that the same output is produced in a piece of translated code as in the original code. Deeply nested constructs are of interest, due to their non-trivial nature.

2. Intent preservation - how the intent of a certain piece of code is preserved after a translation.

3. Construct preservation - how well non-changed constructs are left intact after a back-and-forth translation including and excluding changes to the code base.

Note that unmodified extracts of code from Applewood Computing Accounting System [59] could not be used due to the limited range of language constructs supported by our prototype. Instead, examples containing as much supported code as possible were gathered and the unsupported language constructs were removed.

In evaluating and discussing validity of transformations, we differentiated between two types of validity: *transformation validity* and *specification validity* which are defined as follows:

Definition 4.4.1. Transformation validity *is regarded as guaranteeing that transformations produce a valid result, in both directions, according to specification.*

Definition 4.4.2. Specification validity *is regarded as guaranteeing that relations are specified in a manner such that constructs in one language correctly relates to those in the other language, i.e. the meaning is correct.*

Since transformation validity is largely guaranteed by construct when using a bidirectional transformation language, the main focus the evaluation was specification validity.

5

Survey of Cobol Language Construct Frequency

This chapter presents the tool developed in order to analyze COBOL language constructs' occurrence frequency, and the results obtained by running it against a sample project. Performing the analysis is interesting as the results can be used to scope the work in the remaining result chapters. The validity of generalizing the results are discussed at the end of this chapter.

5.1 The Developed Analysis Tool

In order to measure occurrences of certain COBOL language constructs in a certain code base, the compiler GnuCOBOL [6] (formerly OpenCOBOL) was modified, by adding counting code to the parser generator specification (a formal grammar and AST-construction code is already specified together for use by the parser generators yacc or Bison in the existing source). A total of 429 different grammar rules, that we deemed relevant to distinguish between, are distinctly recognized. This approach allows every program that can be compiled with GnuCOBOL to be analyzed by simply compiling them, using pre-existing build scripts of the examined project.

5.2 Analysis Results

The most commonly used language constructs used in Applewood Computing Accounting System [59] are listed in Table 5.1. In summary, basic constructs such as IF, ADD, MOVE etc. are used in this COBOL source too. The most used constructs were record declarations and PICTURE clauses, similar to struct and variable declarations. VALUE IS and USAGE IS are related to these variable definitions. We also saw that the DISPLAY and ACCEPT commands, responsible for console output and input, were important.

Table 5.1: Summary of the 50 most commonly used language constructs in the COBOL program Applewood Computing Accounting System.

Language construct	Occurrences	Language construct	Occurrences
RECORD declaration	41660	PICTURE declaration	30980
VALUE IS (on PICTURE)	17935	MOVE	11919
CONDITION (88 level)	7934	IF	5287
DISPLAY	4022	CONSTANT ENTRY (78 level)	3301
(DISPLAY) WITH FOREGROUND-COLOR	3288	GOTO	3187
PARAGRAPH declaration	2700	USAGE IS BINARY-LONG SIGNED	2671
PERFORM _	1699	Screen declaration	1581
Screen opt COLUMN	1528	REDEFINES (RECORD)	1417
USAGE IS BINARY-CHAR SIGNED	1233	USAGE IS BINARY-SHORT SIGNED	1215
ADD _ TO _	1153	USAGE IS COMP	1062
WRITE	1050	ELSE	1036
Screen opt LINE	849	(WRITE) BEFORE/AFTER ADVANCING lines	844
(RELEASE/WRITE/REWRITE) FROM	814	Section header declaration	801
USAGE IS COMP-3	793	ACCEPT	776
OCCURS	719	EXIT SECTION	645
(EVALUATE) WHEN	570	(STRING) DELIMITED BY	553
(accept opt) FOREGROUND-COLOR	538	CALL	510
(CALL/ENTRY) USING	509	(accept opt) UPDATE/DEFAULT	435
USAGE IS BINARY-CHAR UNSIGNED	419	CLOSE	416
(STRING/UNSTRING) WITH POINTER	363	Screen opt USING	351
STRING	343	READ	335
(DISPLAY) WITH HIGHLIGHT	323	(DISPLAY) WITH ERASE EOL	311
FILE-CONTROL ASSIGN	296	FILE-CONTROL SELECT	296
FD (file type)	285	(DISPLAY) WITH ERASE EOS	265
SUBTRACT _ FROM _	262	Screen opt FOREGROUND-COLOR	236

5.3 Limitations and Validity of Generalization

A limitation of the developed tool is that it only supports analysis of COBOL programs compilable by GnuCOBOL (including pre-processing). As we only tested the tool on one program, we cannot be certain the results obtained are generalizable. However, we have no indications that they should not generalizable, except that the system under test is a console program, hinting that DISPLAY and ACCEPT might be used differently in more integrated programs.

6

Choosing a Target General-Purpose Language

This chapter explains why we chose C++ as an optimal target language. First, a list of popular languages (relating back to RQ1 that the language should be well-known) is obtained. Second, a comparison between these languages is given. Last, a discussion follows, concluding that only a subset of C++ (a hybrid between C and C++) will be supported by our work, due to time limitations.

6.1 The Comparison

The software quality company Tiobe compiles a list of languages each month, ranking them by popularity [3]. At the time of writing, the most recent one is Tiobe Index for April 2017, which places the following languages, in order, as the most popular ones: Java, C, C++, C#, Python, PHP, VB.Net, and JavaScript. These languages will be used in the comparison. The headers below match the list of criteria introduced in the Method section.

6.1.1 Static or Dynamic Typing

The selected languages can be classified according to the two paradigms of loosely typed scripting languages (Python, PHP, VB.NET, and JavaScript), and typically compiled statically type-checked languages (Java, C#, C, C++, **along with COBOL**). Due to this difference, we decided to remove the scripting languages from the remainder of the comparison and focus on Java, C#, C, and C++.

> For sources supporting the claims made for each respective language, refer to the language specifications for Java [63], C [64], C++ [65], C# [66], and COBOL [67].

6.1.2 Memory Management and Environment

One important difference between Java and C# on one side, and C, C++, and COBOL on the other, is how memory is managed. In COBOL, from what we saw in the Language Construct Survey, most memory is allocated statically at the beginning of a program, and is available there until program execution halts. On function calls, memory may also reside on the stack for parameters and return values. Even though

we did not see it in actual code, it is possible to allocate memory on the heap [68]. Such heap memory must be returned manually to the system. Both C and C++ are similar in this regard. Java, on the other hand, only allows primitive data types to be allocated statically and stored on the stack. More complex objects need to be heap-allocated. Additionally, memory is managed through a garbage collector. C# makes the distinction in the language by heap-allocated and garbage collected classes, and stack-allocated structs.

6.1.3 Primitive Types

The primitive types in C#, C, and C++ allow for storing numbers in signed and unsigned form, compared to Java, which only allows signed numbers (except for `char`, which is 2 byte unsigned). COBOL has a complex type system with pictures (PICs) that store numbers or strings. The numbers can be signed or unsigned, and contain a decimal, which might be implied (not present in memory). The memory representation can be binary (the same as in C/C++/C#/Java), or text-based, so that the number can be printed directly as a string. Additionally, it can be stored as a packed form of the text-based format. All languages support 32 and 64-bit floating precision numbers (typically IEEE-floats).

Unlike the other languages, COBOL strings encode information about formatting, which can be arbitrarily specified by the programmer. A string, for example, might contain three letters followed by three digits, separated by an implied decimal (i.e. a comma not stored in memory at runtime, but still included when formatting the string). This information is stored in the type itself. When outputting these strings to the console, using the DISPLAY command, they are formatted with for example implied decimal printed and leading zeroes converted to spaces [69]. Also, as already hinted at in the COBOL language construct frequency survey, the DISPLAY command can manipulate the console appearance, something not natively supported in the other languages. For example, a majority of all DISPLAY commands in the analyzed code had the option to display WITH FOREGROUND COLOR. These features should be possible to emulate in all the other languages by helper functions.

6.1.4 Classes and Objects

COBOL has the notion of records, where primitive data types and other sub-records are stored together, like a struct in C, C++, or C#. A class in C++, C# and Java is also more or less similar. Of important note are REDEFINES records, where a sub-record aliases another in memory, implemented in C# as field offsets, and similar to a union in C or C++. In Java, such functionality is not directly available to the developer. While C and C++ do not guarantee memory storage location, i.e. where in a struct a member is stored, or using unions to read aliased memory, it is so common to do so anyway that it almost is an informal standard according to our experience. COBOL also supports classes since a recent version, but we have not investigated the matter further, as the language construct survey revealed they were not used in practice (it can also be argued that they are of less practical interest, as the vast majority of existing COBOL code was written before they were introduced

into the language).

6.1.5 Functions

COBOL, C, and C++ allow something similar to a function to be called. Arguments are passed on the stack and returned there (or via processor registers, at the implementation's discretion). Similarly, Java and C# allow methods to be called on objects, even though separate functions are not allowed. Of note again is that whole objects cannot be passed on the stack in Java - a reference to a heap-allocated object must be used. C# and C++ allow operator overloading, where for example arithmetic operators such as '+' and '*' are mapped to user-defined functions. A COBOL function (a *paragraph* in COBOL terminology) may not call itself recursively, in contrast to its C, C++, C# and Java counterparts.

6.1.6 Basic Syntax

The syntax of basic operations and control flow are very similar in C, C#, C++ and Java. The concepts are similar in COBOL too, but COBOL tends to use separate statements for basic arithmetics (ADD, ADD-TO, MULTIPLY etc.) and a separate COMPUTE-statement for more complex arithmetic expressions. In C, C++, C# and Java, everything tends to be an expression, regardless of how complex it is. One important difference in COBOL compared to the other languages is that it can handle integer arithmetic overflow by natively controlling program flow using simple branching (like an if statement). Another important difference is that data can flow from several sources to several targets, compared to C/C++/C#/Java which is more single-data load-store oriented. An example of this is the ADD-statement, which allows several sources to be added together and then stored at several destinations, all in one statement.

6.1.7 Pre-Processing and Meta-Programming

C, C++, and COBOL have pre-processors that allow simple macro expansion. C++ provides advanced meta-programming features through templates, allowing code run at compile-time to decide how a type or function should be constructed. Java and C# have simple template-like semantics implemented as generics, mimicking C++ templates in the sense that a type may take another type as an argument to construct the final instantiation at compile-time. C++ templates, and C++'s ability to reference memory byte-by-byte and bit-by-bit, allows all COBOL data types to be emulated in a bit-perfect manner. Considering that there are theoretically an infinite number of data types in COBOL (as they are all custom), this is an important result. Furthermore, operator overloading in C++ can provide native arithmetic and casting support for these types. Paired together with structs and unions, COBOL records can be emulated directly, as the memory can be forced to align perfectly. As previously mentioned, these custom types can be stored statically, on the stack as function parameters, and be passed as function return values in C++. Finally, the COBOL-equivalent of an array (a `table`) and its access semantics, can be similarly translated using a C++ template implementation.

6.2 Discussion

Given that COBOL data types can be constructed in C++, it seems to be the best choice given our constraints. It is also evident how Java's memory model might interfere with programming, as it is constrained to garbage-collected references in many cases. This might require developers to write code in a very specific manner, not very like how they are used to, making the popularity of the language less relevant. Writing the code to fit Java-to-COBOL translation might also interfere with standard Java compile-time checks about what is allowed or not on regular static types. These reasons might defy the purpose of having a transformation altogether.

Comparing C and C++, the latter is more or less a version of the former with new features. In order to fit the scope of the thesis, we chose to support a special variant of C++ that allows us to use the C++ features for implementing our standard library supporting COBOL data type emulation. User code is, on the other hand, restricted to what transformations are supported (see chapter 8), more like the basic C syntax. This restriction simplifies translation, as templates and overloaded operators need not be supported in the transformations. As already mentioned, classes are not supported either, due to scoping of our work.

7

Emulating COBOL Data Types in C++

This chapter describes the standard library we developed for emulating COBOL data types in C++. The results will be used in chapter 8 when answering research question RQ3, for when a COBOL data type that have no native C++ counterpart is translated.

The standard library was realized with C++ templates and structs with operator overloads in these. Three different data types were realized: one for the default decimal numeric type (based on EBCDIC-encoded strings), one for the computational field type packed decimal, and one for regular strings. We concluded that other types of so-called computational fields could be translated to regular integral types and floating point numbers. COBOL records can be represented with C++ structs, and REDEFINES records with unions, as the developed data types are stored exactly as their equivalent COBOL counterparts, bit by bit. Furthermore, COBOL tables may be represented as C++ arrays, and loop indexing by the native memory indexing type `std::size_t`. The COBOL type FILLER (that occupies dummy memory) is trivially represented by an empty array of bytes. Using these conversions allows every data type representable in COBOL to also be representable in C++.

7.1 Enhanced Byte Arrays

A native array of raw bytes (chars) in C++ cannot be directly passed by-value on the stack [65]. This limitation was circumvented by defining a templatized struct, `ByteArray<L>`, containing an array of the given size, as in Listing 1. The `ByteArray<L>` could then be used to have full control over the memory when defining the emulation types. It should be noted that the compiler is allowed to lay out the memory as it wants [65], but we rely on that it will order fields in the order they are defined in a struct (in our experience, this is the common case). Furthermore, we rely on that the compiler does not try to align the memory addresses (e.g. on 32-bit or 64-bit boundaries), a feature that might be needed to be turned off in some implementations.

```
typedef unsigned char byte;

template<std::size_t L> struct ByteArray {
    private:
        byte data[L];

    public:
        ByteArray() : data() {}

        byte& operator[](std::size_t idx) {
            return data[idx];
        }

        const byte operator[](std::size_t idx) const {
            return data[idx];
        }
};
```

Listing 1: An array of raw bytes that can be passed by-value on the stack.

7.2 The Default Decmial Type

The default COBOL data type allows numbers to be stored in the EBCDIC format (similar to an ASCII string, but another encoding) [67]. The four least significant bits contain the number (0-9), and the four most significant ones are normally set high (1111). The data might be signed, where the sign mask replaces the four most significant bits on either the first or the last byte; which alternative is used can be configured by code when declaring the data type. Finally, the number may contain decimals, and the decimal sign (a dot) may be *implied*, meaning that it is not explicitly stored as a byte in the data. Implied decimal is the default. The byte-width of the integer and decimal part is configured in code when declaring the data type.

The emulation type was realized as a templatized struct, shown in Listing 2, which allows all mentioned parameters to be specified into the compile-time type. Of important note is how the expression defining the byte width can be resolved at compile-time and baked into the instantiated type. Overloaded operators on the type, allowing addition and add-assign as examples of arithmetic operators, as well as casting to strings and doubles, are also of interest. To enhance readability, the constructors that instantiate specific instances to zero, a given default value, or a copy of the packed decimal type discussed below (its internals are accessed by making it a `friend`), are not shown.

7.3 Packed Decimal

COBOL also allows packing a default decimal number into almost half the bit-size, by removing the four most significant bits, thus storing two digits into the same byte [67]. Given that the resulting type cannot be directly printed, the decimal character is always implied, and thus not stored. In order to store the sign, some extra four bits are required, at the least significant position. For an odd number of digits, the sign bits can be baked into the type. For an even number of digits, the four most significant bits on the first byte are set to zero, and the four least significant bits on the last byte contain the sign. It also happens to be the case that four sign bits are reserved even for unsigned numbers, in which case they are set to all ones.

The packed decimal emulation type (see Listing 3) was realized similarly to the unpacked version. Its internal implementation temporarily converts to the unpacked version for calculations, and then back (optimizing for code simplicity over performance). While the cast overloading technically can handle all cases, C++ does not automatically use it e.g. when adding a packed decimal to another packed decimal. Therefore, the operators are overloaded in this type too. When adding a packed decimal to the default unpacked version, casting is performed and the arithmetic operation is handled directly in the default type.

7.4 Strings

COBOL also allows its data types to take an arbitrary form of text and numbers, possibly formatted in a certain way [67] (see chapter 6 for details). Thus, the formatting is part of the type information. As a data type that contains text cannot be used directly in numerical calculations (without converting it first, e.g. by using REDEFINES records and overlaying a decimal type padded by FILLER), they are represented by a separate string type. The multi-byte string type Double-Byte Character Set (DBCS) is not explicitly supported by our implementation. However, the implementation should be able to handle such data transparently, but there is no way of specifying literals of such type in C++, and we have not developed any value-converter as a workaround.

The string type was realized using a template that poses as a tuple, allowing different pieces of the string with different formatting to be concatenated together. For example, the code `PIC<X<5>, C, Z<3>, V, N<2>> WS_HELLO("Hello,00314");` defines:

- five characters of any type (X),
- followed by a comma (C),
- followed by three digits, where leading zeroes should be replaced with space when formatting the string (Z),
- followed by an implied decimal that should not be stored in memory, but still printed when formatting the string for output (V),

- and finally followed by two regular digits (no special formatting of leading digits) (N).

Listing 4 and 5 shows how the types were implemented in more detail. The tuple is recursively defined and accessed using tail recursion, with inspiration from the method presented by [70], but using types instead of literals. Note how each type (X, C, Z, V, N etc.) has a cast-to-string operator overload and occupies as much memory as needed. Note also how the V option is only part of the compile-time type, as it has no members and thus occupies no memory at run time.

7.5 Summary

We have presented one way to represent COBOL data types in C++. Relating back to the feature goals in the Method section, trivial tests shown that all listed features work. The storage is bit-perfectly stored in byte arrays. Overloads on the encapsulating integer types allow simple arithmetic operations (with overflow detection) and casting between formats. String types can be formatted properly. COBOL RECORDS and REDEFINES can be handled by native language constructs (C++ struct and union), given some assumptions on the compiler, notably outside the language specification but as commonly implemented by vendors. The results obtained can thus be used when answering research question RQ3.

```
template<std::size_t Integers,/* Template parameters allowing */
        std::size_t Decimals, /* arbitrary width.             */
        bool Sign,/* Signed or unsigned? */
        bool ImpliedDecimal,/* Skip decimal point in data representation? */
        bool SignFirst>/* Store sign first or last? */
struct FormattedNumber {
    friend PackedFormattedNumber<Integers, Decimals, Sign>;

    public:
    /* The data type's width is configured by a constexpr, consisting of
       calculations on template parameters resolvable at compile-time */
    typedef ByteArray<Integers + Decimals + (ImpliedDecimal ? 0 : 1)> data_t;
    data_t data;

    /* Constructors and helper code removed for brevity. */

    FormattedNumber& operator+=(const FormattedNumber& rhs) {
        byte carry = 0;
        for(std::size_t i = Integers + Decimals /*- 1 + 1*/; i-- > 0; ) {
            /* A simple ripple carry adder, removed for brevity. */
        }

        if(carry > 0) {
            /* This allows raising an overflow signal,
               e.g. implemented as an exception. */
        }

        return *this;
    }

    FormattedNumber operator+(const FormattedNumber& rhs) {
        FormattedNumber lhs(this);
        lhs += rhs;
        return lhs;
    }

    operator std::string() const {
        /* Body removed for brevity. Allows casting to string. */
    }

    operator double() const {
        /* Body removed for brevity. Allows casting to double. */
    }
};
```

Listing 2: Some parts of the code emulating a default COBOL number.

```
template<std::size_t Integers,
        std::size_t Decimals,
        bool Sign>
struct PackedFormattedNumber {
    friend FormattedNumber<Integers, Decimals, Sign>;

    typedef ByteArray<(Integers + Decimals + 1 + 1)/2> data_t;
    data_t data;

    template<bool ImpliedDecimal, bool SignFirst>
    PackedFormattedNumber(
            const FormattedNumber<Integers, Decimals, Sign, ImpliedDecimal,
            SignFirst>& unpacked) {
        /* Copy constructor taking an unpacked number. Removed for brevity. */
    }

    template<bool ImpliedDecimal, bool SignFirst>
    operator FormattedNumber<Integers, Decimals, Sign, ImpliedDecimal,
            SignFirst>() const {
        /* The cast operator overload allows unpacking. */
        return FormattedNumber<Integers, Decimals, Sign, ImpliedDecimal,
            SignFirst>(*this);
    }

    /* More constructors and helpers removed for brevity. */

    PackedFormattedNumber& operator+=(const PackedFormattedNumber& rhs) {
        /* Unpack, perform, re-pack; condensed and abbreviated. */
        FormattedNumber<Integers, Decimals, Sign> unpacked(*this);
        FormattedNumber<Integers, Decimals, Sign> rhsUnpacked(rhs);
        unpacked += rhsUnpacked;
        PackedFormattedNumber<Integers, Decimals, Sign> packed(unpacked);
    }

    PackedFormattedNumber operator+(const PackedFormattedNumber& rhs) {
        PackedFormattedNumber lhs(*this);
        lhs += rhs;
        return lhs;
    }

    operator std::string() const {
        return static_cast<FormattedNumber<Integers, Decimals, Sign>>(*this);
    }

    operator double() const {
        return static_cast<FormattedNumber<Integers, Decimals, Sign>>(*this);
    }
};
```

Listing 3: Some parts of the code emulating a packed COBOL number.

```
template<std::size_t Repeat = 1> // Alphanumeric, cobol X, C++ X
class X {
    protected:
    char data[Repeat];

    public:
    X() : data() {}

    X(const char* input) {
        memcpy(&data, input, Repeat);
    }

    X(const char input) : data{input} { } // Means first input, rest 0 (if any)

    operator std::string() const {
        return std::string(data, Repeat);
    } };

template<std::size_t Repeat = 1> using A = X<Repeat>; // Alphab., cobol A, C++ A
template<std::size_t Repeat = 1> using N = X<Repeat>; // Numeric, cobol 9, C++ N

class D : public X<1> { // Decimal dot, cobol ., C++ D
    public: D() : X<1>('.') {} };

class C : public X<1> { // Comma, cobol ,, C++ C
    public: C() : X<1>(',') {} };

class V {// Implied decimal, cobol V, C++ V
    public: operator std::string() const {
        return std::string(",");
    } };

template<std::size_t Repeat = 1>// Trim leading zeroes, cobol Z, C++ Z
class Z : public X<Repeat> {
    public:
    operator std::string() const {
        std::string trimmed(this->data, Repeat);

        // Find leading zeroes, but skip the last if it contains all zero.
        std::size_t toFill = std::min(trimmed.find_first_not_of('0'),
                trimmed.size() - 1);

        // Trim them off
        trimmed.erase(0, toFill);

        // Fill up and concatenate
        return std::string(toFill, ' ') + trimmed;
    } };
```

Listing 4: Code that defines string types and their formatting.

```cpp
/* This is the base case that halts the recursion. */
template<typename... Tail>
class PIC {
    public:
    // Halt recursion
    PIC(const char* input) { }
    void display() const {
        std::cout << std::endl;
    }
};

/* This is the recursive case, that takes one string type off the list,
   and then passes the others to the
   next stage using a parent constructor. */
template<typename Head, typename... Tail>
class PIC<Head, Tail...> : PIC<Tail...> {
    Head head;

    public:
    /* Traverse the input string by adding
       sizeof(Head) to the next stage. */
    PIC(const char* input) : PIC<Tail...>(input + sizeof(Head)), head() {
        /* Copy the part of the input string to the memory occupied. */
        memcpy(&head, input, sizeof(Head));
    }

    void display() const {
        /* Print head, call tail, that will recursively
           print its head etc. */
        std::cout << static_cast<std::string>(head);
        PIC<Tail...>::display();
    }
};
```

Listing 5: Code that implements the tuple and printing of it.

8

Creating a Model-Driven Source-to-Source Compiler

This result chapter describes the developed prototype source-to-source compiler that takes COBOL code, generates C++ code (according to the subset of C++ discussed in Chapter 6), and allows changes to the generated C++ code to propagate back to the original COBOL code base. Figure 8.1 shows an overview of the transformation architecture, which is built like a chain, where different bidirectional transformations handle change propagation between code in COBOL and the C++-like language. The translation process consists of 4 steps:

1. **COBOL code to COBOL model**. An Xtext-based COBOL grammar, along with a generated corresponding metamodel, is used in a text-to-model transformation where COBOL code is transformed to a COBOL model (and vice versa). Xtext is used to perform the text-to-model transformations.

2. **COBOL model to intermediate model**. QVT-R transformations transform between a COBOL model and the intermediate model based on common *concepts* found in programming languages (and vice versa).

3. **Intermediate model to C++ model**. QVT-R transformations transform between the intermediate model and a C++ model.

4. **C++ model to C++ code**. An Xtext-based C++ grammar is used in a text-to-model transformation, where a C++ model is transformed to C++ code (and vice versa).

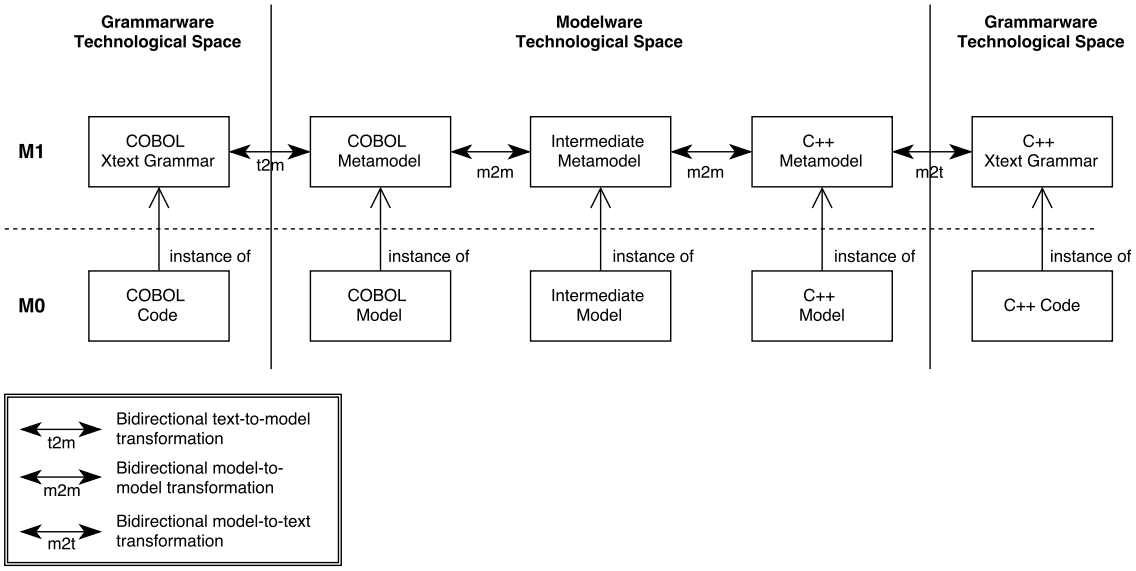

Figure 8.1: An illustration of the architecture used to implement the source-to-source compiler. Note that transformations are defined on the M1 level and operates with instances in the M0 layer as input.

The remainder of this chapter will describe the approaches devised and the resulting implementation for realizing the architecture in Figure 8.1. Section 8.1 describes the outcome of the grammar conversion to Xtext grammars, used for the text-to-model transformations (steps 1 and 4). Section 8.2 describes our notion of *concepts* used in the intermediate model. Section 8.3 describes our proposed method for defining concepts, and the resulting intermediate model as well as the relations between concepts and language constructs in both languages. Section Section 8.4 describes the approaches devised for implementing QVT-R transformations, as well as examples of the resulting transformations. Finally, in Section 8.5 the resulting source-to-source compiler is evaluated based on representative examples.

8.1 Transformations between Code and Model

Two Xtext grammars, one for COBOL and one for C++, allowed the implementation of model-to-text transformations between a piece of code and its corresponding model, in both directions. The whole of Lämmel and Verhoef's COBOL grammar [60], with a few exemptions, was converted to the Xtext format according to the strategy outlined in Section 4.4.1. Two types of issues were encountered and solved. The first issue was handling the ambiguity in the COBOL grammar. Xtext has a backtrack option which can resolve ambiguities at runtime. However, enabling this option led to that the generated parser code contained methods which exceeded Java's method size limit [71]. Therefore, we dealt with ambiguities by refactoring and inserting syntactic predicates (dictating which option to take, when there is an ambiguous choice) where needed. Second, due to the way ANTLR 3 tokenizes input, many lexer rules in the original COBOL grammar [60] had to be rewritten into a combination of data rules and lexer rules. Listing 8.1 shows an example of this.

If given the input 'A1', both `ALPHABETIC_USER_DEFINED_WORD` and `COBOL_WORD` will match. However, since `ALPHABETIC_USER_DEFINED_WORD` is defined first, it will always be matched, meaning that `COBOL_WORD` will never be matched in this case. For the grammar to be valid, this must not be the case. Therefore, a data rule `COBOLWord` must be created, which matches either a `ALPHABETIC_USER_DEFINED_WORD` or a `COBOL_WORD`. This data rule can then be used by other rules, ensuring that the original intent of the grammar is captured.

Listing 8.1: An example of lexer rules that have to be re-written as a combination of data rules and lexer rules. The example is an extract from our Xtext COBOL grammar.

```
COBOLWord: ALPHABETIC_USER_DEFINED_WORD | COBOL_WORD;

terminal ALPHABETIC_USER_DEFINED_WORD: (DIGIT (DIGIT|DASH_OR_BACKSLASH)*)?
  ALPHA (ALPHA|DIGIT)* (DASH_OR_BACKSLASH+ (ALPHA|DIGIT)+)*;
terminal COBOL_WORD: (ALPHA|DIGIT)+ (DASH_OR_BACKSLASH+ (ALPHA|DIGIT)+)*;

terminal fragment DASH_OR_BACKSLASH: '\\'|'-';
terminal fragment ALPHA: 'A'..'Z'|'a'..'z';
terminal fragment DIGIT: '0'..'9';
```

The conversion of the C++ grammar [62] did not suffer from the problems described above. Backtracking could be enabled and the grammar functioned properly. The metamodels, generated by Xtext, for respective grammar needed only one type of modification: marking classes, corresponding to parser rules that serve as delegators, as abstract. For example, the `Statement` rule in Listing 8.2 is only delegating to other rules, which become subclasses to the corresponding `Statement` class generated by Xtext. Therefore, the `Statement` class had to be marked as abstract.

Listing 8.2: An example of a parser rule which serves as an delegator to other rules. The example is an extract from our Xtext C++ grammar.

```
Statement : LabeledStatement | CompoundStatement | ExpressionStatement
        | IfStatement | SwitchStatement | WhileStatement
    | DoWhileStatement | ForStatement | JumpStatement;
```

8.2 Information in Code

Code conveys at least two types of information: one related to coding style, another related to the actual intent of the code, i.e. the language constructs used. When transforming code, these different aspects are reflected in different parts of the transformation. It is desirable that the intent is transferred to the new code base, but the coding style is unimportant. On the contrary, it is desirable that the coding style is preserved when back-transforming code, but the original intent is not important as it should change to the new intent of any new changes. In order to limit the scope of this thesis, preserving coding style was deliberately excluded from the research questions, and was thus not further explored. Instead, focus was put on transforming the intent programmers have when writing code.

When back-transforming code, it might be desirable if the original intent, to the greatest extent possible, is preserved in parts that have not changed (this might also be the case with the local coding style used in the files changed, but as already

mentioned, preserving coding style is outside the scope of this thesis). Albrecht et al. [52], Yellin [55] and Krieg-Brückner's [53] methods of transforming code might encounter problems with transferring intent between code bases. High-level language constructs might be translated to low-level language constructs. For example, a for loop used to (intended to) iterate over a collection might be translated into a GOTO-construction, as it is the general translation for loops. However, if the target language has a special foreach-construct to iterate over a collection, the intent could have been captured but the information instead becomes lost.

8.2.1 Concepts

When considering how to transform intent between different code bases (recall RQ3d), we must first have a clear understanding of how broad the term is, thus answering the question of how different two approaches can be and still have the same intent. Recall RQ3b, stipulating that the transformed models should behave the same in terms of output, given the same input. Applied to the domain of programming, we argue that for example a for-loop and a while-loop can encode the same intent: looping through a collection. We thus propose defining and using *concepts*, defined as:

Definition 8.2.1. Concept *a grouping of language constructs (at least one in each language) which encode the same human intent (Definition 1.2.1) about what the computer should do. As such, a concept has a set of concrete representations in the form of language constructs and all of these representations are considered interchangeable since they encode the same intent.*

Note the similarity with Krieg-Brückner's definition of concept [53], but also the extension to make it even more general, by raising the level of abstraction from language construct level, to the intent of the programmer (e.g. iterating through a collection, regardless of how the iteration was technically accomplished in code). Thus, the *concepts* exist on the same level of abstraction as *computational thinking*, as defined by Wing [54].

To illustrate the notion of a concept, two concepts are exemplified in Figrue 8.2 and Figure 8.3. The example in Figrue 8.2 shows a concept for incrementing a variable. As can be seen, the concept has one concrete representation in COBOL, and three concrete representations in C++; note that two of these have a *representational condition*, expressing that the increment needs to be exactly 1 for the representations to be valid. We claim that the same intent is represented by any of these concrete representations. In Figure 8.3, a numeric expression assignment concept is illustrated. There is one concrete representation in C++, whereas there are 5 in COBOL; note that 4 of the representations in COBOL require that the expression (`expr`) only consists of their respective operator (e.g. +, -). We claim that these concrete representations present the same intent. The interesting thing to note is that an increment can indeed be represented as expression, i.e. `i += x` produces the same result as the expression `i = i + x`. However, these two operations are still represented as two different concepts, because we believe they encode different intent. If a programmer writes `i = i + x`, instead of `i += x`, there is

likely a reason behind that choice, i.e. a special intent. This could be, for example, to mimic a mathematical formula. Therefore, they are kept separate.

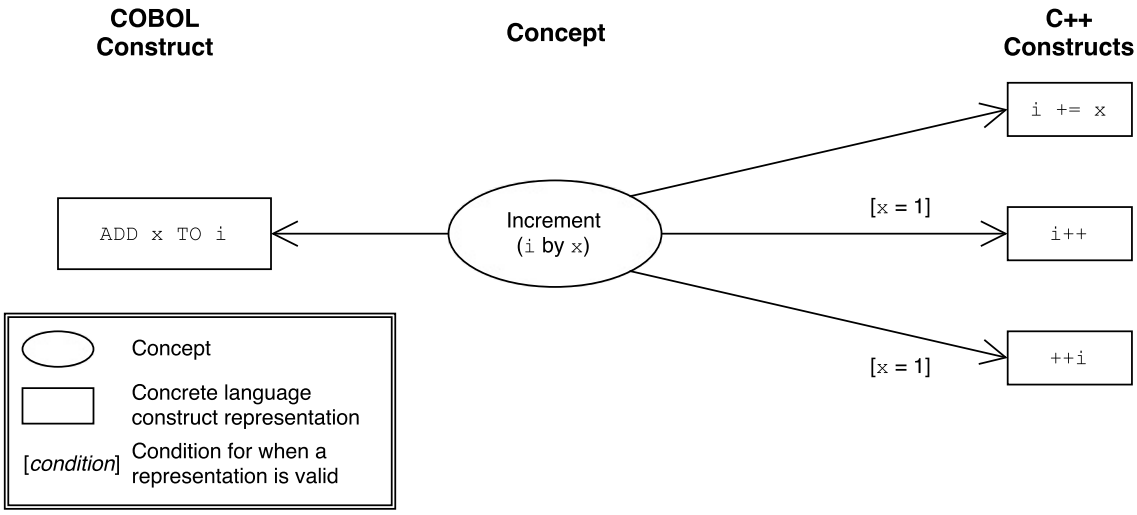

Figure 8.2: An illustration of an Increment concept, where a variable `i` is incremented by `x` (a constant or a variable). The concept has one concrete representation in COBOL (left) and three in C++ (right).

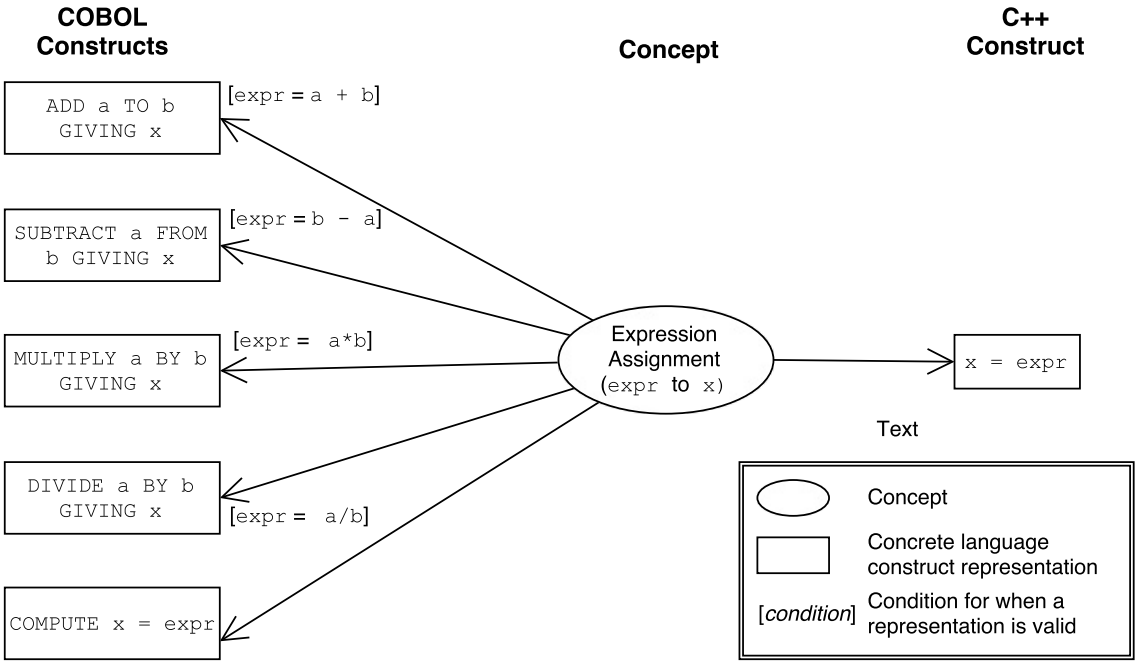

Figure 8.3: An illustration of a numeric Expression Assignment concept, where an arbitrary numeric expression (`expr`) is assigned to a variable `x`. The concept has five concrete representation in COBOL (left) and one in C++ (right).

Before defining the intermediate model, which models concepts and serves as an example implementation adapted to our specific requirements, concerning a COBOL to C++ language transformation, we first motivate the need for a separate model

altogether. It can be argued that the concepts could be captured in the transformation itself. However, writing such transformations between the language models directly becomes cumbersome as the number of combinations grows in quadratic order. For example: translating between a for, while, and do-while loop in one language to for, foreach, and while in another, makes for 9 transformations between them (the number of edges in a complete bipartite graph, i.e. $m * n$ where m is the amount of representation in one language, and n in the other). With an intermediary model, the number of transformations grows linearly with the number of supported language constructs, making for only 6 transformations in the example. Writing more transformations by hand is not only more tedious, but also more error prone. Therefore, the intermediary model also facilitates extension to more languages. While we only have two languages, we anticipate that our results become easier to generalize by supporting extension from the start. These reasons motivate our use of an intermediary model.

8.3 The Intermediate Concept Model

This section outlines the definition of the intermediate concept model, including the specific concepts we defined, and the relations to each language. While the content of this section is quite general, it has to be considered as a case relevant to COBOL and C++, thus being less generalizable than the rest of our approach. Other languages might require other concepts to capture their specific traits. Furthermore, there is more than one way to define concepts. Our choices are rather personal and reflect what we consider to be *a programmer's intent* in code.

The general approach used for defining the canonical form of concepts is a modification of Greatest Common Divisor (GCD) (presented by Yellin [55]), namely *GCD over concepts*. That is, we define a model representation of a concept as the GDC between all different concrete representations of that concept in both languages, such that there is a many-to-one mapping between the different representations and the concept. In addition, we consider whether a concept available at a high abstraction level in one language, but not the other, can be emulated in a standard library in the other language. As such, a higher GCD is created. For this to be the case, the emulated concept must be comprehensible and easy to use.

Note that translation is performed over concepts, meaning that every translatable entity in both languages must be represented, under their representational condition, by exactly one concept in the intermediate model, i.e there must not be a many-to-many mapping between constructs and concepts, only a many-to-one mapping. Furthermore, the granularity of a concept determines the extent to which specific intent is preserved; if a concept is very general, it may represent different intents which in the translation are considered as the same since any representation of the concept is valid. Consider again the increment concept illustrated in Figure 8.2. Had an increment been represented by the more general expression concept (Figure 8.3), the intent of specifying a special increment operation (shorthand operation) may not be preserved after a translation, as `i = i + x` is also a valid translation. It should also be noted that our approach does not involve any translation between concepts, only over concepts, in order to preserve intent.

The remainder of this section describes the different concept classes defined in the intermediate model.

8.3.1 Program

The concept of a sequential program is similarly represented in both languages, namely an entity which has a name, a set of variables, and a set of statements to be executed. Due to time limitations, we consider a program as one file, meaning that any external dependencies are disallowed. Furthermore, we do not consider any way of preserving scopes in C++, since that concept does not exist in COBOL. Therefore, all variables are global within the program. Figure 8.4 shows a diagram of the Program class and its related classes in the intermediate concept model.

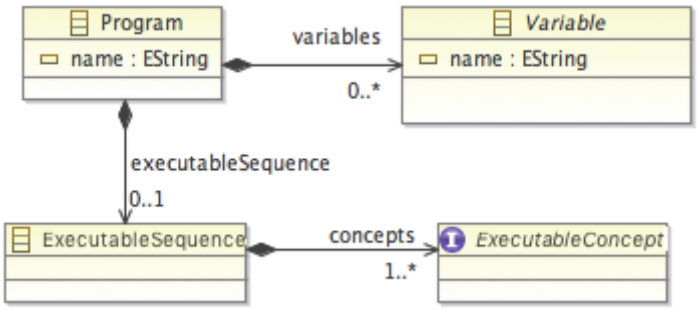

Figure 8.4: UML diagram of the representation of a program, its variables, and its executable sequence of concepts (statements)

8.3.2 Variables

As presented in Chapter 6, COBOL's data types are of a special nature. There is a straightforward translation of two COBOL types to C++, namely when USAGE is COMP-1 (single precision float) or COMP-2 (double precision float). COBOL's other data types cannot be easily expressed using native C++ data types. Therefore, COBOL data types were emulated in C++, as described in Chapter 7. The resulting representation of a variable in the intermediate concept model is a model-centric representation of COBOL's numeric data types, as illustrated in Figure 8.5. The relations between each language and the variable concepts can be found in Table 8.1 and Table 8.2.

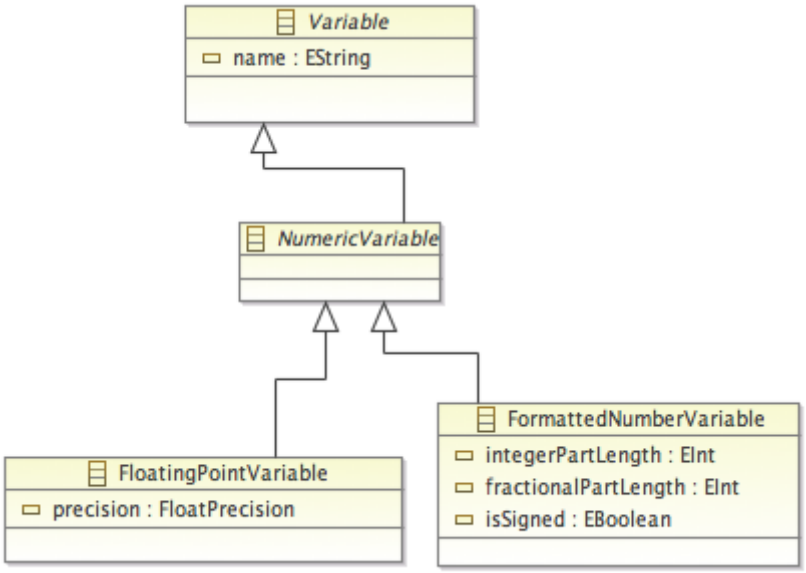

Figure 8.5: UML diagram of classes representing variables.

COBOL	Concept	Comment
77 *name* USAGE COMP-1 VALUE *val*	FloatingPointVariable [name=*name*, precision=SINGLE, defValue=*val*]	Note: initialization value is optional and a dash is replaced by an underline character
77 *name* USAGE COMP-2 VALUE *val*	FloatingPointVariable [name=*name*, precision=DOUBLE, defValue=*val*]	Note: initialization value is optional and a dash is replaced by an underline character
77 *name* PIC 9(x).9(y) VALUE *val*	FormattedNumberVariable [name=*name*, intPartLen=x, fractionalPartLen=y, defValue=*val*]	If the integer or fractional part is 0, then it may be left out. An S at the beginning of the PIC clause means it is signed. Note: initialization value is optional and a dash is replaced by an underline character

Table 8.1: The relations between COBOL variables types and variable concepts. (Note that some details are excluded to enhance readability.)

C++	Concept	Comment
float *name* = *val*;	FloatingPointVariable [name=*name*, precision=SINGLE, defValue=*val*]	Note: initialization value is optional and a dash is replaced by an underline character
double *name* = *val*;	FloatingPointVariable [name=*name*, precision=DOUBLE, defValue=*val*]	Note: initialization value is optional and a dash is replaced by an underline character
FormattedNumber<*i*,*f*> *name* = *val*	FormattedNumberVariable [name=*name*, intPartLen=*i*, fractionalPartLen=*f*, defValue=*val*]	FormattedNumber is an emulated COBOL data type. Note: initialization value is optional and a dash is replaced by an underline character

Table 8.2: The relations between C++ variables types and variable concepts. (Note that some details are excluded to enhance readability.)

8.3.3 Arithmetic Expression-Assignments

COBOL has multiple options for calculating expressions and assigning their numerical outcome to variables. We categorize these into two kinds: those statements that only involve one operator (e.g. `ADD`, `MULTIPLY`), and those that involve a combination of operators (i.e. the `COMPUTE` statement). Note that all single-operator statements can also be expressed as a `COMPUTE` statement. All single-operator statements have a shorthand for assigning a value to a variable that also acts as an operand in the expression. For example, `ADD 1 TO WS-I`, is the shorthand for `ADD 1 TO WS-I GIVING WS-I`.

In C++, there is mainly one statement for calculating expressions and assigning their numerical outcome to variables, which is the '=' statement. Similarly to COBOL, there also exist shorthand operators for assigning a value to a variable which acts as an operand in the expression. For example, `i += 1` is equivalent to `i++` or `++i`, all of which are shorthand notations for `i = i + 1`. C++ allows an expression with different operators to be used in a shorthand notation, e.g. `i += 4 - 3 * 2`, which is not possible to express as a shorthand notation in COBOL.

Since both COBOL and C++ give the possibility to express shorthand notation, we defined a concept `NumericalSelfAssignment` to represent these and keep the intent of the shorthand notations (see Figure 8.6). Note the GCD for the shorthand notation results in that `NumericalSelfAssignment` only has one operator (COBOL limitation) and only one target variable (C++ limitation); in COBOL one shorthand statement can act on multiple variables. For example, `ADD 1 TO WS-I WS-J` would be translated to two statements `i += 1; j +=1;` in C++ since it cannot be represented as one. To represent any other numerical expression, not expressed in shorthand notation, we devised a concept `ExpressionAssignment` which assigns

a general expression to a variable. The relations between arithmetic expression-assignments in both languages and the devised concepts can be found in Table 8.3 and Table 8.4

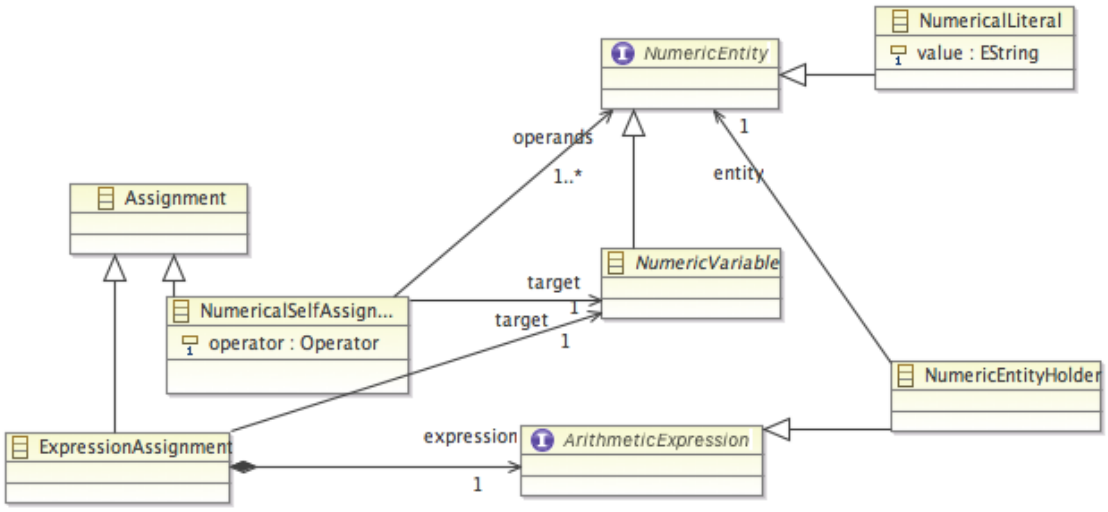

Figure 8.6: UML diagram of classes representing expressional assignments.

COBOL	Concept	Comment
ADD a b TO c	NumericalSelfAssignment [operand=+, operators={a,b}, target=c]	Note: having more than one target (e.g. TO c d) will result in one NumericalSelfAssignment per target.
ADD a TO b GIVING c d	ExpressionAssignment [expression=a+b, targets={c,d}]	**Condition**: expression may only contain + (plus) operands.
SUBTRACT a b FROM c	NumericalSelfAssignment [operand=-, operators={a,b}, target=c]	Note: having more than one target (e.g. TO c d) will result in one NumericalSelfAssignment per target.
SUBTRACT a FROM b GIVING c d	ExpressionAssignment [expression=a+b, targets={c,d}]	**Condition**: expression may only contain - (minus) operands.
MULTIPLY a BY b	NumericalSelfAssignment [operand=*, operators={a}, target=b]	Note: having more than one target (e.g. BY b c) will result in one NumericalSelfAssignment per target.
MULTIPLY a BY b GIVING c d	ExpressionAssignment [expression=a*b, targets={c,d}]	**Condition**: expression may only contain one * (multiply) operand.
DIVIDE a INTO b	NumericalSelfAssignment [operand=/, operators={a}, target=b]	Note: having more than one target (e.g. INTO b c) will result in one NumericalSelfAssignment per target.
DIVIDE a (BY/INTO) b GIVING c d	ExpressionAssignment [expression=a/b, targets={c,d}]	**Condition**: expression may only contain one / (divide) operand.
COMPUTE a b = *expr*	ExpressionAssignment [expression=*expr*, targets={a,b}]	**Condition**: *expr* must not consist of a single value
MOVE a TO b c	ValueAssignment [value=*a*, targets={b,c}]	-

Table 8.3: The relations between COBOL arithmetic assignment operations and their corresponding concepts. (Note that some details are excluded to enhance readability.)

C++	Concept	Comment
c += a + b;	NumericalSelfAssignment [operand=+, operators={a,b}, target=c]	**Condition**: the right hand side expression may only contain + (plus) operands
a += *expr*;	ExpressionAssignment [expression= a + *expr*, targets={a}]	**Condition**: the relation above must not be matched
c -= a + b;	NumericalSelfAssignment [operand=-, operators={a,b}, target=c]	**Condition**: the right hand side expression may only contain + (plus) operands
a -= *expr*;	ExpressionAssignment [expression= a - *expr*, targets={a}]	**Condition**: the relation above must not be matched
b *= a;	NumericalSelfAssignment [operand=*, operators={a}, target=b]	**Condition**: the right hand may only be a value, not an expression
a *= *expr*;	ExpressionAssignment [expression= a * *expr*, targets={a}]	**Condition**: the relation above must not be matched
b /= a;	NumericalSelfAssignment [operand=/, operators={a}, target=b]	**Condition**: the right hand may only be a value, not an expression
a /= *expr*;	ExpressionAssignment [expression= a / *expr*, targets={a}]	**Condition**: the relation above must not be matched
a = b = *numericalExpression*;	ExpressionAssignment [expression=*numericalExpression*, targets={a,b}]	-

Table 8.4: The relations between C++ arithmetic assignment operations and their corresponding concepts. (Note that some details are excluded to enhance readability.)

8.3.4 Conditional Branching

Both COBOL and C++ use the if-else conditional statement to alter the execution flow of a program. There is a straightforward one-to-one relation between the if-else constructs of both languages since there are no other features (e.g. ifelse) available in neither language. As such, we devised a concept `ConditionalBranching` for representing this feature (See Figure 8.7). The relations between if-statements in both languages and the devised concept `ConditionalBranching` can be found in Table 8.5 and Table 8.6

Both COBOL and C++ also have means for comparing numerical expressions, means to negate conditions, and the common boolean logical operators (e.g. logical AND, logical OR). As such, they become the GCD for a `Condition` (See Figure

8.7). However, C++ also has multiple bit-level operations for manipulating bits. This concept does not at all exist in COBOL and although these could be emulated in COBOL, it would not make sense to do so since the intent of our source-to-source compiler is to further development of legacy code. Any bitwise operation emulation code would already have been written in COBOL, if such data were to be handled by existing code. As such, we do not support translation of bit-level operations.

Another difference between COBOL and C++ regarding conditions is that C++ regards any number that is not zero as being of the boolean value: true. Therefore, an expression such as `if(i)`, where `i` is an `int`, is allowed. This is not the case in COBOL, and therefore it is not part of the GCD of Condition. Instead, this has to be represented as an `ArithmeticCondition` (e.g. `i != 0`) in the concept model.

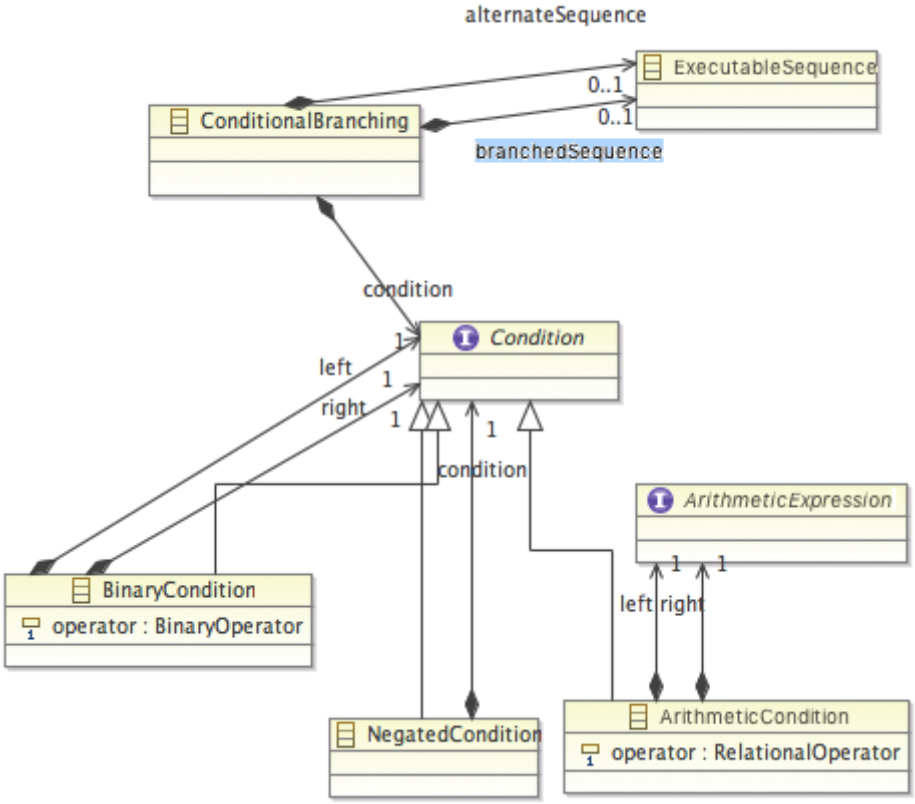

Figure 8.7: UML diagram of classes representing the ConditionalBranching concept and Condition.

COBOL	Concept	Comment
IF *cond* THEN *statements1* ELSE *statements2*	ConditionalBranching [condition=*cond*, branchedSequence=*statements1*, alternateSequence=*statements2*]	*statements1* and *statements2* are transformed into concepts before assignment.

Table 8.5: The relation between COBOL if-statement and the ConditionalBranching concept. (Note that some details are excluded to enhance readability.)

C++	Concept	Comment
if (*cond*) { *statements1* } else { *statements2* }	ConditionalBranching [condition=*cond*, branchedSequence=*statements1*, alternateSequence=*statements2*]	*statements1* and *statements2* are transformed into concepts before assignment.

Table 8.6: The relation between C++ if-statement and the ConditionalBranching concept. (Note that some details are excluded to enhance readability.)

8.3.5 Loops

In both languages, there are multiple ways of defining loops. In COBOL, the primary construct for defining loops is the `PERFORM` statement. The `PERFORM` statement has 3 different formats for defining loops. First, `PERFORM TIMES`, which executes the body of the loop n times. Second, `PERFORM UNTIL`, which executes the body of the loop until a given condition becomes true. Third, `PERFORM VARYING`, which executes the body of the loop, varying an index variable by a fixed step (e.g. increments by one each iteration), until a given condition becomes true.

In C++, there are mainly three different constructs for expressing loops: `while`, `do while`, and `for`. `while` and `do while` execute the body of the loop until a condition becomes false, with the difference that `while` checks the condition before the loop body is entered. `for` also executes the body of the loop until a condition becomes false, but it has an explicit structure which requires one statement that is to be executed before entering the loop, and one statement that is to be executed at the end of each iteration (although these can be empty statements).

To accommodate the different loop constructs, we defined two concepts in the intermediate model: `ConditionalLooping` and `ConsecutiveIteration`, as illustrated in Figure 8.8. `ConditionalLooping` is a concept that represents a basic loop which executes the body of the loop until a condition becomes false. Every loop construct within both languages is expressible as a `ConditionalLooping` concept

(in combination with extra variables). In addition to `ConditionalLooping`, we defined `ConsecutiveIteration` which represents an iteration with an index that is consecutively increased/decreased by a fixed step until a condition becomes false. As such, `ConsecutiveIteration` is a subclass of `ConditionalLooping`, adding an index variable. The reason for including a separate iteration concept is twofold: to simplify transformations, since the complexity would otherwise increase, and to highlight that there is a difference in concept; for example, iteration over a collection is often represented differently (e.g. in a for or foreach loop) than a loop accepting input from the user (e.g. a while loop). The relations between the looping constructs in both languages and their corresponding concepts are outlined in Table 8.7 and 8.8.

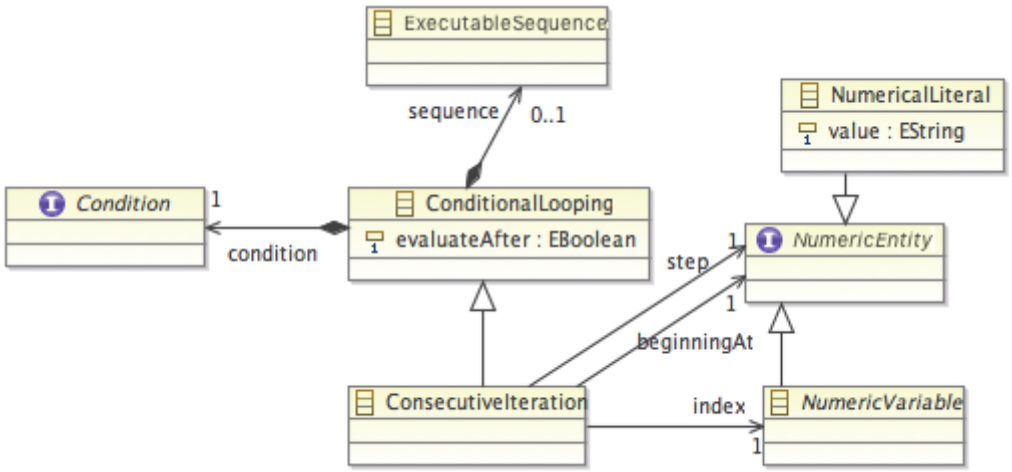

Figure 8.8: UML diagram of classes representing the loop concepts.

COBOL	Concept	Comment
PERFORM n TIMES *statements*	ConsecutiveIteration [index=i, step=1, beginningAt=0, condition=i<n, sequence=*statements*]	**Condition**: there may be no other concept referencing the index variable within the iteration. An index variable needs to be created dynamically when translating from COBOL. *statements* are transformed into concepts before assignment. Note: there are theoretically infinitely many combinations of `beginningAt`, `step`, and `condition`, which will yield the same amount of iterations n
PERFORM VARYING i FROM b BY s UNTIL *cond* *statements*	ConsecutiveIteration [index=i, step=s, beginningAt=b, condition=*cond*, sequence=*statements*]	*statements* are transformed into concepts before assignment.
PERFORM UNTIL *cond* *statements*	ConsecutiveIteration [index=i, step=s, beginningAt=b, condition=*cond*, sequence=*statements*]	**Condition**: there is an increment/decrement statement with step s of an index variable i last amongst *statements*. i is assigned the value b is and not used in any other statements between the loop and the assignment. *statements* are transformed into concepts before assignment.
PERFORM UNTIL *cond* *statements*	ConditionalLoop [condition=*cond*, sequence=*statements*]	**Condition**: the relation above must not be matched *statements* are transformed into concepts before assignment.

Table 8.7: The relations between looping statements in COBOL and their corresponding concepts. (Note that some details are excluded to enhance readability.)

C++	Concept	Comment
for (i = b; $cond$; i += s) { $statements$ }	ConsecutiveIteration [index=i, step=s, beginningAt=b, condition=$cond$, sequence=$statements$]	Note: i++ and ++i are equivalent of i+=1. $statements$ are transformed into concepts before assignment.
while ($cond$) { $statements$ }	ConditionalLooping [condition=$cond$, sequence=$statements$]	$statements$ are transformed into concepts before assignment.
do { $statements$ } while ($cond$);	ConditionalLooping [condition=$cond$, sequence=$statements$, evaluateAfter=false]	$statements$ are transformed into concepts before assignment.

Table 8.8: The relations between looping statements in C++ and their corresponding concepts. (Note that some details are excluded to enhance readability.)

8.3.6 Printing

In both languages, there are multiple ways to print data to a console (and other outputs). COBOL's way of printing, the `DISPLAY` statement, natively supports many options, e.g. to change background color, and print in specific positions. [69] C++'s way of printing, using the standard `cout`, is more limited. [65] However, there exist C++ libraries for achieving the same prints as in COBOL. Due to time limitations, we only supported one option: whether to print with a new line or not. This print concept is displayed in Figure 8.9. The relations between the print constructs in both languages and their corresponding concept are outlined in Table 8.9 and 8.10.

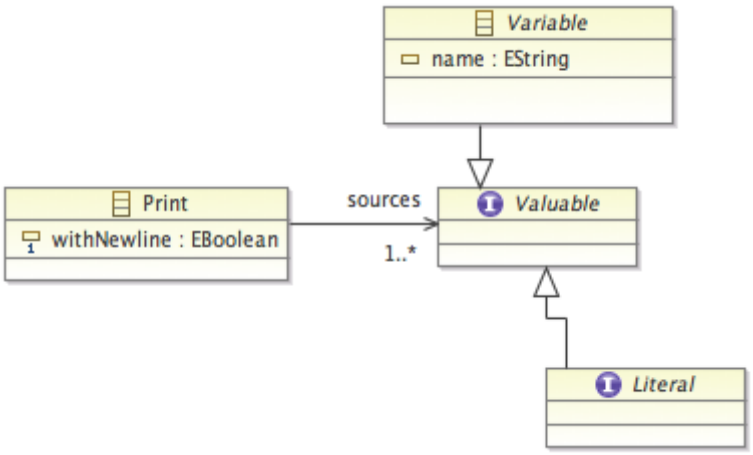

Figure 8.9: UML diagram of the Print concept and related classes.

COBOL	Concept	Comment
DISPLAY *a b*	Print [sources={*a*, *b*}, withNewline=true]	Note: the presence of the option `WITH NO ADVANCING` sets `withNewline` to false

Table 8.9: The relation between the `DISPLAY` statement in COBOL and the corresponding Print concept. (Note that some details are excluded to enhance readability.)

C++	Concept	Comment
cout « *a* « *b* « endl;	Print [sources={*a*, *b*}, withNewline=true]	Note: the presence of `« endl` sets `withNewline` to true, otherwise if would be false.

Table 8.10: The relation between the `cout` statement in C++ and the corresponding Print concept. (Note that some details are excluded to enhance readability.)

8.4 Transformation between Concept Model and Language Models

This section will present the approaches used when implementing the bidirectional transformations, between the models, in QVT-R. Examples will be provided to illustrate the approaches. Based on our research questions, we formulated four requirements for a transformation between two source models M_{COBOL} and M_{C++}, giving rise to the programs P_{COBOL} and P_{C++}, to be considered valid if:

R1 Given input I, P_{COBOL} and P_{C++} produce the same output O.

R2 The order of any executable sequence of statements, $s_{COBOL} \in P_{COBOL}$, is maintained for the corresponding executable sequence of statements, $s_{C++} \in P_{C++}$.

R3 The order of the declared variables in P_{COBOL} is the same as the order of the declared variables in P_{C++}.

R4 No additional statements that do not contribute to represent the same semantic meaning are introduced.

8.4.1 Basic Relations

As explained in the beginning of this chapter, a transformation $M_{COBOL} \leftrightarrow M_{C++}$ between the models M_{COBOL} and M_{C++} is broken down into two separate transformations involving the intermediate model: $M_{COBOL} \leftrightarrow M_I$ and $M_I \leftrightarrow M_{C++}$. Each of these transformations have a top relation (the main entry point) that maps the concept of the highest entity transformed, a program. These top relations delegate to other inter-dependent relations, each specifying the relation between one concept and its concrete representations (in M_{COBOL} or M_{C++}). In the case of $M_I \leftrightarrow M_{C++}$, this top relation specifies that for every Translation Unit in C++, there should exist one Program in the intermediate model, as can be seen in Listing 8.3. As can also be seen in Listing 8.3, the relation requires that another relation `mainFunctionRelation`, which is responsible for transforming the content of a program, holds, and through it, the rest of the transformation takes place.

Listing 8.3: The top relation *translationUnit2program*, mapping a Translation Unit to an intermediate Program

```
top relation translationUnit2program {
  enforce domain c traUnit : c::TranslationUnit {
    externalDeclarations = declaration:ExternalDeclaration {}
  };
  enforce domain inter intermediateProgram:Program {
    variables = variable : intermediate::Variable {}
  };

  when { declaration2Variable(declaration, variable);}
  where { mainFunctionRelation(traUnit, intermediateProgram);}
}
```

Similarly to `translationUnit2program`, a relation `sourceProgram2program` (displayed in Listing 8.4) specifies that for every COBOL source program, named n, there should exist one Program in the intermediate model, named n. Every COBOL source program requires there to be an identification division, whereby `sourceProgram2program` requires there to exist an IdentificationDivisionContent. The actual content of the program is transformed by `statementListRelation`.

Listing 8.4: The top relation *sourceProgram2program*, mapping a COBOL Source Program to an intermediate Program

```
top relation sourceProgram2program {
  n : String;
  enforce domain cob cobProg:COBOLSourceProgram {
    identificationDivisionContent =
      cobol::IdentificationDivisionContent {},
    dataDivision = dd:DataDivision {
      workingStorageSection = ws:WorkingStorageSection {
          dataDescriptionEntries =
            desc:DataDescriptionEntryRegular {}
        }
      },
    name = n
  };

  enforce domain inter interProg:Program {
    variables = variable : intermediate::Variable {},
    name = n
  };

  when {variable2Variable(desc, variable);}
  where { statementListRelation(cobProg, interProg);}
```

```
}
```

Both `translationUnit2program` and `sourceProgram2program` depend on their respective relation `declaration2Variable` and `variable2Variable` being realized. This is necessary due to the fact that global variables, which may be referenced arbitrarily in the transformation process, must be available to be referenced during the entire transformation.

In general, many relations are of a similar straightforward nature as `translationUnit2program` and `sourceProgram2program`, i.e. mapping language constructs to a concept in the intermediate model (note that several language constructs can be composed into one concept, but never the other way around)[1]. However, there are some additional general patterns and exceptions, not covered in these two relations, which will be explained and exemplified throughout the remainder of this section.

8.4.2 Delegated Relations

In some situations, it is favorable to split up a relation into multiple relations that transform the same object. We call such relations *delegated relations* and we identified three use cases where they might be favorable to use:

UC1 When a property of a class, which is part of the relation, can have multiple options and the class has several properties.

UC2 When a property of a class, which is part of the relation, may be undefined or empty and the class has several properties.

UC3 When the type of a property of a class, which is part of the relation, is also present in other classes.

Consider the example in Listing 8.5, where an arithmetic binary operation in the COBOL model is mapped to an arithmetic binary operation in the intermediate model. The subject (`ArithmeticBinaryOperation`) has a property (operator) which may have multiple options (e.g. +, -, *), and the subject has other properties (left and right expressions). Therefore, the operator property is delegated to other relations, one for each possible option of the property (e.g. `plusToPlus` and `minusToMinus`), by stating them in the `where` clause of the relation (`arithmeticBinaryOperationRelation`). As such, the delegated relations take care of transforming the operator property. By combining the delegated relations with the operator `xor` in the `where` clause, exactly one of them has to be true for the relation to be true. Comparing this approach to that of Listing 8.6 (not using delegated relations), where the entire relation is specified multiple times with only the operator being different, delegated relations results in less code duplication.

Listing 8.5: An example of using delegated relations when an object's property can have multiple options

```
relation arithmeticBinaryOperationRelation {
  enforce domain cob cOP : cobol::ArithmeticBinaryOperation {
```

[1]For the full specification, refer to https://github.com/sebbe33/mde-cobol-c-transpiler. Note that all relations are not implemented to the full extent as specified in 8.3

```
    left = cobLeft : cobol::ArithmeticExpression {},
    right = cobRight : cobol::ArithmeticExpression {}
  };

  enforce domain inter iOP : intermediate::ArithmeticBinaryOperation {
    left = interLeft : intermediate::ArithmeticExpression {},
    right = interRight : intermediate::ArithmeticExpression {}
  };

  where {
    plusToPlus(cOP, iOP) xor minusToMinus(cOP, iOP)
      xor multiplyToMultiply(cOP, iOP) xor divideToDivide(cOP, iOP)
      xor powerToPower(cOP, iOP);
    arithmeticExpressionRelation(cobLeft, interLeft);
    arithmeticExpressionRelation(cobRight, interRight);
  }
}

relation plusToPlus {
  enforce domain cob cob : cobol::ArithmeticBinaryOperation { operator = '+' };
  enforce domain inter inter : intermediate::ArithmeticBinaryOperation {
    operator = intermediate::ArithmeticBinaryOperator::PLUS
  };
}
relation minusToMinus { ... }
...
```

Listing 8.6: An example of not using delegated relations when an object's property can have multiple options

```
relation arithmeticBinaryOperationPlus {
  enforce domain cob cOP : cobol::ArithmeticBinaryOperation {
    operator = operator = '+'
    ... -- other properties
  };

  enforce domain inter iOP : intermediate::ArithmeticBinaryOperation {
    operator = intermediate::ArithmeticBinaryOperator::PLUS
    ... -- other properties
  };

  where {
    arithmeticExpressionRelation(cobLeft, interLeft);
    arithmeticExpressionRelation(cobRight, interRight);
  }
}

relation arithmeticBinaryOperationMinus {
  enforce domain cob cOP : cobol::ArithmeticBinaryOperation {
    operator = operator = '-'
    ... -- other properties
  };

  enforce domain inter iOP : intermediate::ArithmeticBinaryOperation {
    operator = intermediate::ArithmeticBinaryOperator::MIUNS
    ... -- other properties
  };

  where { ... }
}

...
```

Use case UC2 for delegated relations is a special case of use case number 1 (a property can be regarded as having the value `undefined` as one option and any other value as another option). Consider the relation specified in Listing 8.7, specifying the mapping between a Compound Statement in C++ and an executable sequence in

the intermediate model; for each `CompoundStatement` with a `StatementList` there should exist an `ExecutableSequence`, where the statements in the `StatementList` and the `ExecutableSequence` corresponds. The `statementList` property is allowed to be undefined (e.g. in the case of an empty if-statement body), and an empty `ExecutableSequence` should in such cases be created. However, due to the way QVT-R performs pattern matching, a `CompoundStatement` with an undefined `statementList` property will not be matched in `compoundStatementToSequence`, since the relation requires a `CompoundStatement` to have a defined `statementList` property.

A solution to the problem presented above, using delegated relations, is presented in Listing 8.8. `compoundStatementToSequenc` has been converted to only handle the undefined case and delegates the non-empty case to `compoundStatement ListToSequence` (notice that the we only require `compoundStatementListToSequence` to be invoked if the `statementList` is defined, by use of implication).

Listing 8.7: An example of not using delegated relations when an object's property can be empty

```
relation compoundStatementToSequence {
  enforce domain c cmpStat : c::CompoundStatement {
    statementList = statements : c::StatementList {}
  };

  enforce domain inter seq : intermediate::ExecutableSequence {};
  where { statementsToStatements(statements, seq); }
}
```

Listing 8.8: An example of using delegated relations when an object's property can be empty

```
relation compoundStatementListToSequence {
  enforce domain c cmpStat : c::CompoundStatement {
    statementList = statements : c::StatementList {}
  };

  enforce domain inter seq : intermediate::ExecutableSequence {};
  where { statementsToStatements(statements, seq); }
}

relation compoundStatementToSequence {
  enforce domain c cmpStat : c::CompoundStatement {};
  enforce domain inter seq : intermediate::ExecutableSequence {};
  where { (not cmpStat.statementList.oclIsUndefined()) implies
    compoundStatementListToSequence(cmpStat, seq);
  }
}
```

Use case UC3 for delegated relations is when the translation of properties can be generalized amongst several relations. Consider the example in Listing 8.9. Both `binaryConditionRelation` and `negatedCondition` require that their nested conditions (all of the type `Condition`) are transformed. Therefore, a `conditionRelation` can be specified as a delegated relation which handles the transformation between `Conditions`. In fact, in this case, this definition is recursive, since `conditionRelation` delegates back to `binaryConditionRelation` and `negatedCondition`. For this delegation to work in Medini QVT, the `Condition` class must be abstract, since otherwise an instance of type `Condition` would be created in `binaryConditionRelation`/ `negatedCondition`, and the delegation in `conditionRelation` would return false since it would not match any of the specified relations.

Listing 8.9: An example of using delegated relations for generalizing relations for shared property types

```
relation binaryConditionRelation {
  enforce domain cob c1 : cobol::BinaryCondition {
    left = cobLeft : cobol::Condition{},
    right = cobRight : cobol::Condition{}
  };
  enforce domain inter c2 : intermediate::BinaryCondition {
    left = interLeft : intermediate::Condition{},
    right = interRight : intermediate::Condition{}
  };
  where {
    conditionRelation(cobLeft, interLeft);
    conditionRelation(cobRight, interRight);
  }
}

relation negatedCondition {
  enforce domain cob c1 : cobol::NegatedSimpleConditions {
    condition = nestedCobCond : cobol::Condition {}
  };
  enforce domain inter c2 : intermediate::NegatedCondition {
    condition = nestedInterCond : intermediate::Condition {}
  };
  where { conditionRelation(nestedCobCond, nestedInterCond); }
}

relation conditionRelation {
  enforce domain cob cobCond : cobol::Condition {};
  enforce domain inter interCond : intermediate::Condition {};
  where {
    binaryConditionRelation(cobCond, interCond)
    xor arithmeticExpressionConditionRelation(cobCond, interCond)
    xor negatedCondition(cobCond, interCond);
  }
}
```

8.4.3 Enforcing Order

In order to ensure Requirement R2 and R3, stating that the order of corresponding statements and variables should remain the same after a transformation, certain measures have to be taken. The QVT-R standard does not, to our best interpretation, specify exactly in which order relations should be executed, other than if a relation is dependent on another relation (i.e. has a reference to another relation in its `where` clause) whereby that other relation must be executed first. In Medini QVT, this might present a problem when transforming ordered collections (e.g. sequences, ordered sets).

Consider the relations in Listing 8.10, mapping COBOL statements to executable concepts in the intermediate model. Clearly, the order of the statements should remain the same when transformed, as per requirement R2. However, given a sequence of COBOL statements `statementList = Sequence{IfStatement{}, ComputeStatement{}, IfStatement{}}`, the corresponding executable concepts in the intermediate model will become `executableConcepts = Sequence{ConditionalBranching, ConditionalBranching, ExpressionAssignment}`. The reason for this is that Medini QVT executes the entire `ifToCondBran` relation first since it is specified before `moveToValAssign`, meaning that all statements in `statementList` matching the relation (even if they are not in order) will be transformed. Then

`moveToValAssign` is executed.

Listing 8.10: An example illustrating non-enforcement of order when stating relations between ordered collections

```
relation statementListToExecSequence {
  enforce domain cob statList : cobol::StatementList {};
  enforce domain inter execSeq : cobol::ExecutableSequence {};
  where {
    ifToConditionalBranching(statList, execSeq);
    moveToValAssign(statList, execSeq);
  }
}

relation ifToCondBran {
  enforce domain cob statList : cobol::StatementList {
    statements = cobol::IfStatement { ... }
  };
  enforce domain inter execSeq : cobol::ExecutableSequence {
    concepts = intermediate::ConditionalBranching {...}
  };
  where { ... }
}

relation moveToValAssign {
  enforce domain cob statList : cobol::StatementList {
    statements = cobol::MoveStatement { ... }
  };
  enforce domain inter execSeq : cobol::ExecutableSequence {
    concepts = intermediate::ValueAssignment {...}
  };
  where { ... }
}
```

To avoid the problem described above, we found that the relations mapping the different types of statements must act on `Statement` and `Concept` rather than `StatementList` and `ExecutableSequence` (see Listing 8.11). As such these relations act on an atomic element in a collection, and therefore they cannot affect the order. Instead, the order is determined by the `statementListToExecSequence` relation. Given that `Statement` and `Concept` are abstract, the pattern matching will match any element in the collections in `statementListToExecSequence`. Hence, the transformation will execute element by element in the collections, not skipping any element due to its type, and the transformation of the concrete type (e.g. `IfStatement`/`ConditionalBranching`) will be delegated to the matching relation (e.g. `ifToCondBran`).

Listing 8.11: An example illustrating enforcing order when stating relations between ordered collections

```
relation statementListToExecSequence {
  enforce domain cob statList : cobol::StatementList {
    statements = s : cobol::Statement {}
  };
  enforce domain inter execSeq : cobol::ExecutableSequence {
    concepts = c : intermediate::Concept {}
  };
  where {
    ifToCondBran(s, c);
    moveToValAssign(s, c);
  }
}

relation ifToCondBran {
  enforce domain cob ifst : cobol::IfStatement {...};
```

```
  enforce domain inter cb : intermediate::ConditionalBranching {...};
  where { ... }
}

relation moveToValAssign {
  enforce domain cob ms : cobol::MoveStatement { ... };
  enforce domain inter va : intermediate::ValueAssignment {...};
  where { ... }
}
```

Although the above approach solves the order problem, we found that it has the disadvantage that there must be a one-to-one mapping between elements in the collections in terms of the *amount* of elements produced (note that the relation can still be ambiguous). The approach in Listing 8.10 has the advantage that, for example, the `moveToValAssign` relation may map one `MoveStatement` to several `ValueAssignments` since it has access to the collection. Given that `moveToValAssign` in Listing 8.11 works on an atomic element of a collection, and not the collection itself, it can only transform that element to one corresponding element; if it was to create more elements, they would simply not be added to the collection. For example, the COBOL statement `ADD 1 TO i j` gives the equivalent two statements in C++ `i += 1` and `j += 1`. As such, there is a a one-to-one mapping between the amount statements since one COBOL statement is represented by several C++ statements. The difference between this relation and other ambiguous relations between constructs and concepts is that there is a one-to-many mapping of instances, whereas the other relations are of a one-choose-one-of-many nature, meaning that one alternative must be chosen (not all). We have not been able to produce a solution which maintains order and is able to handle both these types of relations.

8.4.4 Dealing With Strings in Medini QVT

To manipulate and read data in QVT-R, OCL operations are used. The 2.0 OCL specification [72], which Medini QVT implements, has a limited set of standard operations for the dealing with the data type String. There are no methods for directly converting an Integer to a String, removing, accessing, or checking conditions on characters (although some of these are available in 2.0+ OCL specifications). These operations are necessary for transforming a COBOL PICTURE string.

The native data type Sequence (a subtype to Collection) provides operations for accessing, removing, and checking condition on individual elements. Therefore, to implement these operations for the String data type, we defined two queries: one of which converts a String to a Sequence of Strings (there is no character type in OCL and as such a character has to be represented by a String) and one in the opposite direction, both found in Listing 8.12. As such, a String can be converted to a Sequence of strings and operations such as `at(index)`, `forAll(condition)`, `includes(element)`, and `collect(condition)` become available. If needed, the Sequence of Strings can then be converted back to a String. An example of usage can be found in Listing 8.13 where three queries, used by relations dealing with variable transformations, are defined.

The operation converting an Integer to a String was also defined as a query, using a recursive digit-by-digit approach, as illustrated in Listing 8.12 (notice that

a modulo operation had to be defined since the one defined by Medini did not work properly).

Listing 8.12: The defines queries for converting between String and Sequence of strings, and an intToString operation

```
query stringToSequence (s : String) : Sequence(String) {
  Sequence{1..s.size()}->collect(i | s.substring(i, i))
}

query sequenceToString (ss : Sequence(String)) : String {
  if ss->size() = 0 then '' else
    if ss->size() = 1 then  ss->first()
    else ss->first() + sequenceToString( ss->subSequence(2, ss->size()) )
    endif
  endif
}

query modulo10(i : Integer) : Integer { i - ((i/10).floor() * 10) }

query intToString(i : Integer) : String {
  if i >= 10 then
    intToString( (i/10).floor() ) +
    Sequence{'0','1','2','3','4','5','6','7','8','9'}->at( modulo10(i) + 1 )
  else
    Sequence{'0','1','2','3','4','5','6','7','8','9'}->at( i + 1 )
  endif
}
```

Listing 8.13: The listing defines three queries, dealing with COBOL PICTURE strings, used by relations dealing with variable transformations

```
-- checks whether the specified format is signed
query isPICSigned(picString : String) : Boolean {
  let ss = stringToSequence(picString) in ss->includes('s') or ss->includes('S')
}

-- Checks whether a PIC format represents a valid numeric variable
query validNumberDefinition(picString : String) : Boolean {
  stringToSequence(picString)->forAll(s |
    isCharacterNum(s) or s = ',' or s = 'S' or s =  '(' or s = ')')
}

-- Generates a numeric PIC format with specified amount of digits
query createNumericPICFormat(amountOfDigits : Integer) : String {
  -- if more than 4 digits, we use the shorthand 9(x) representation
  if amountOfDigits > 4 then '9(' + intToString(amountOfDigits) + ')'
  else sequenceToString(Sequence{1..amountOfDigits}->collect(i | '9')) endif
}
```

8.4.5 Implementing Alternate Relations

Several of the relations presented throughout Section 8.3 are alternate relations specifying the different concrete representations of a concept, when there are multiple. To implement such a relation between a concept and its concrete representations, we defined each alternate relation as a one-to-one mapping between the concept and a language construct. These relations were then joined in the `where` clause of a composite relation, specifying the many-to-one relation between language constructs and their corresponding concept.

Consider the example in Listing 8.14. Here, the two COBOL constructs `PERFORM VARYING` and `PERFORM TIMES` are mapped to the intermediate concept `ConsecutiveIteration`

(an iteration with an index variable that is being incremented or decremented at constant steps). There are two one-to-one mappings between the concept and `PERFORM VARYING` and `PERFORM TIMES`, respectively (notice that `PERFORM VARYING` is always transformable to `ConsecutiveIteration`, whereas `PERFORM TIMES` is only transformable under certain conditions, omitted here for the sake of readability). Then, there is the `consecutiveIterationRelation` composite relation, which delegates the task of transforming either `PERFORM` statement to a `Consecutive Iteration` (and vice versa). As such, a many-to-one mapping for the concept is implemented.

Listing 8.14: An example of implementing alternate relations. The example shows the mapping of COBOL constructs to a ConsecutiveIteration

```
relation performVaryingInline2ConsecutiveIteration {
  enforce domain cob perform:PerformStatement {
    performInline = performInline : PerformInline {
      performVaryingPhrase = untilPhrase : PerformVaryingPhrase {...}
    },
    ...
  };
  enforce domain inter iteration : ConsecutiveIteration {...};
  where { ... }
}

relation performTimes2ConsecutiveIteration {
  enforce domain cob perform:PerformStatement {
    performInline = performInline : PerformInline {
      performTimes = times : PerformTimes {...}
    }
  };
  enforce domain inter iteration:intermediate::ConsecutiveIteration {...};
  when { /* Conditions for determining when valid */ }
}

-- This relation gets called when transforming an iteration
relation consecutiveIterationRelation {
  enforce domain cob stat : cobol::Statement {};
  enforce domain inter iter : ConsecutiveIteration {};
  where {
    performTimes2ConsecutiveIteration(stat, iter) or
    performVaryingInline2ConsecutiveIteration(stat, iter);
  }
}
```

Looking at the relations in 8.14 it is trivial to see that a transformation in the direction $M_{COBOL} \rightarrow M_{Itermediate}$ will be valid: calling on `consecutiveIterationRelation` with a `PerformStatement`, containing either a `PerformVaryingPhrase` or a `PerformTimes`, will match maximum one of `performTimes2ConsecutiveIteration` or `performVarying Inline2ConsecutiveIteration` in its `where` clause. Thereof, a corresponding `Consecutive Iteration` concept will be produced if the `PerformStatement` is a valid candidate transformation. In the direction $M_{COBOL} \leftarrow M_{Intermediate}$, by analogy, calling on `consecutiveIterationRelation` with a `ConsecutiveIteration` concept could match both `performTimes2ConsecutiveIteration` and `performVaryingInline2 ConsecutiveIteration` simultaneously, due to the nature of the `or` operator. However, this does not compromise validity, since only one statement will be returned from the `consecutive IterationRelation` (recall the explanation for this in Section 8.4.3). That is, depending on the implementation of the QVT-R interpreter, two statements may be instantiated, but only one will be used; the other one will be disregarded and will not be part of M_{COBOL}.

The choice of which option to select in a one-to-many scenario with this pattern is decided by the QVT-R interpreter. With Medini QVT, it seems like the last relation to be executed, i.e. the last in the chain of 'or' statements, will be selected. Consequently, performing a back-and-forth translation between COBOL and C++, even though nothing has changed, might result in a different model. With Echo, however, theoretically the option which adheres to its least-change principle (see Section 2.4.2) should be selected.

8.4.6 Conditional Relations

Many of the alternate relations presented throughout Section 8.3 have a condition under which they are valid. Such a condition is realized in the `when` clause of a relation. Furthermore, such a condition is often directional, meaning it applies in only one direction. For example, to elaborate on the relation between the `PERFORM TIMES` construct and the `ConsecutiveIteration` concept first presented in Listing 8.14 (refer to the full implementation of its condition in Listing 8.15): The `PERFORM TIMES` statement in COBOL is s a basic loop, executed n times. It has no index variable or any condition which needs to be fulfilled. Therefore, in order to translate a `ConsecutiveIteration`, which might represent a `for` loop in C++, there can be no references to its index variable in its executable sequence. Furthermore, its stop condition must be finite; hence, it must include the index variable, and the stop condition must (for the sake of simplicity) be an `ArithmeticCondition` with the $<$ operator. These conditions are realized in the `when` clause of `performTimes2ConsecutiveIteration`, when transforming in the direction $M_{COBOL} \leftarrow M_{Intermediate}$. Notice that when transforming in the opposite direction, these conditions are not valid. However, when doing so, an index variable is required. Therefore, in the `when` clause, a relation is used to create a unique index variable.

Listing 8.15: An example of a relation which has a condition that needs to be satisfied for the relation to be valid.

```
relation performTimes2ConsecutiveIteration {
  enforce domain cob perform:PerformStatement {
    performInline = performInline : PerformInline {
      performTimes = times : PerformTimes {
        value = timesVal : cobol::ValueSource{}
      },
      statementList = cobStat : cobol::StatementList{}
    }
  };
  enforce domain inter iteration:intermediate::ConsecutiveIteration {
    sequence = execSeq:ExecutableSequence {},
    step = lit : intermediate::NumericalLiteral { value = '1' },
    beginningAt = start : intermediate::NumericalLiteral { value = '0' },
    condition = runCond : intermediate::ArithmeticCondition {
      operator = intermediate::RelationalOperator::LESS,
        left = lNum : NumericEntityHolder { },
        right = rNum : NumericEntityHolder {
          entity = rNumEntity : intermediate::NumericEntity{}
        }
      },
      index = indexVar : NumericVariable { name = 'i' } -- should be generalized
  };
  when {
    -- the directional condition, when going from intermediate -> COBOL
```

```
    if not iteration.index.oclIsUndefined() then
      runCond.left.oclIsTypeOf(NumericEntityHolder) and
      -- index must be included in stop cond.
      runCond.left.oclAsType(NumericEntityHolder).entity = iteration.index and
      not hasReferencesTo(iteration.sequence, iteration.index)
    else true endif;
  }

  where {
    -- relate the times to run literal/variable
    numericLiteralToNumericLiteral(timesVal, rNumEntity)
      or identifierToNumericVariable(timesVal, rNumEntity);
    -- unidirectional relation: create variable if transforming cobol -> inter
    if not perform.oclIsUndefined() then
      -- should be optimized search for existing variables
      createVariable(intermediate::Program.allInstances()->any(true), lNum)
    else true endif;
    statementListToExecSequence(cobStat, execSeq);
  }
}

query hasReferencesTo(seq : ExecutableSequence, v : NumericVariable) : Boolean {
    /* recursively checks if there are any reference to the the variable
    in the sequence, and any of its subsequences */
}

relation createVariable {
  enforce domain inter intermediateProgram:Program {
    variables = fnv : FormattedNumberVariable {
      -- a name should be generated automatically,
      -- but due to time limits we just used 'i'
      integerPartLength = 18, name = 'i'
    }
  };

  enforce domain inter lNum : NumericEntityHolder { entity = fnv};
}
```

The approach presented above is a case where the bidirectional specification capabilities of QVT-R are not enough to gain a desired outcome. Instead, we had to resort to defining part of the transformation, i.e. creating a variable, in a unidirectional manner.

8.5 Evaluation

In this section, our evaluation of implemented source-to-source compiler prototype is presented. The evaluation treats the three aspects of correctness, intent preservation, and construct preservation; each presented in its own subsection.

8.5.1 Correctness

In the following, we discuss the correctness of the developed prototype, based on two examples. The first example is artificial and serves the purpose to cover translation of all implemented language constructs. The second example is a modified extract from Applewood Computing Accounting System [59] (used in the language construct survey in Chapter 5) and should illustrate the applicability to industrial code in context of the supported language constructs.

The first example is shown in Table 8.11. Note that the code has no real meaning other than to illustrate the correctness of translation of all supported language

constructs. It should be fairly trivial to see correctness, based on the relations presented in Section 8.3. The second example is shown in Listing 8.12. The extract shows code related to invoice and account management. As can be seen, both programs behave the same at run-time in terms of output (given the same input) - relating back to research question RQ3b. It should be trivial to see that this is the case due to its simplicity.

COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID. example.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 singlepoint USAGE COMP-1 VALUE 3.1.
77 doublepoint USAGE COMP-2 VALUE 3.1.
77 a PIC 9(12) VALUE 12345678.
77 b PIC 9(12) VALUE 12345678.
77 c PIC 9(12) VALUE 12345678.
77 d PIC 9(12) VALUE 12345678.

PROCEDURE DIVISION .
ADD a TO c
ADD b TO a GIVING c
SUBTRACT a FROM c
MULTIPLY a BY b
DIVIDE a INTO b
COMPUTE a = c * d + b

IF a GREATER 0
  ADD a TO b
  IF 1 GREATER 0
    ADD 1 TO b
  ELSE
    ADD 2 TO b
ELSE
  SUBTRACT a FROM b
END-IF

PERFORM 10 TIMES
   ADD 3 TO a
   SUBTRACT 1 FROM b
END-PERFORM

PERFORM VARYING a
  FROM 4
  BY 2
  UNTIL a > 11
    MULTIPLY c BY b
END-PERFORM

MOVE 0 TO a.
PERFORM UNTIL a > 100
  ADD 3 TO b
  ADD 2 TO a
END-PERFORM

SUBTRACT 100 FROM a
PERFORM UNTIL a > 100
  DISPLAY a
  ADD 2 TO a
  ADD a TO b
END-PERFORM .
```

C++

```
float singlepoint = 3.1f;
double doublepoint = 3.1;
FormattedNumber<12> a(12345678);
FormattedNumber<12> b(12345678);
FormattedNumber<12> c(12345678);
FormattedNumber<12> d(12345678);
FormattedNumber<18> i; // generated

void main() {
  c += a;
  c = b + a;
  c -= a;
  b *= a;
  b /= a;
  a = c * d + b;

  if (a > 0) {
    b += a;
    if (1 > 0) {
      b++;
    } else {
      b += 2;
    }
  } else {
    b -= a;
  }

  for (i = 0 ; i < 10; i++) {
    a += 3;
    b--;
  }

  for (a = 4; a <= 11; a += 2) {
    b *= c;
  }

  a = 0;
  for (a = 0; a <= 100; a+= 2) {
    b += 3;
  }

  a -= 100;
  while (a <= 100) {
    cout << a << endl;
    a += 2;
    b += a;
  }
}
```

Table 8.11: The COBOL program above (left) is intended to show supported language constructs in our prototype. The C++ program (right) is the translated version of the COBOL program (left).

COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID. sl080.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 work-1     PIC s9(8)V99 VALUE 0.
77 amount-out PIC s9(8)V99 VALUE 0.
77 inv-amount PIC s9(8)V99 VALUE 0.

77 bal-0  PIC s9(8)V99 VALUE 0.
77 bal-30 PIC s9(8)V99 VALUE 0.
77 bal-60 PIC s9(8)V99 VALUE 0.
77 bal-90 PIC s9(8)V99 VALUE 0.
77 bal-t  PIC s9(8)V99 VALUE 0.

77 oi-type       PIC 9.
77 oi-net        PIC s9(7)V99.
77 oi-extra      PIC s9(7)V99.
77 oi-carriage   PIC s9(7)V99.
77 oi-discount   PIC s9(7)V99.
77 oi-vat        PIC s9(7)V99.
77 oi-e-vat      PIC s9(7)V99.
77 oi-c-vat      PIC s9(7)V99.
77 oi-paid       PIC s9(7)V99.
77 oi-deduct-amt PIC s999V99.
77 oi-deduct-vat PIC s999V99.

PROCEDURE DIVISION .
COMPUTE inv-amount = oi-net + oi-extra +
   oi-carriage + oi-vat + oi-discount +
   oi-e-vat + oi-deduct-amt +
   oi-deduct-vat + oi-c-vat

IF oi-type = 3
   MULTIPLY -1 BY oi-paid
END-IF
IF oi-type = 2 OR oi-type = 3
   COMPUTE amount-out =
    inv-amount - oi-paid
END-IF

IF  work-1  <  30
 ADD  amount-out  TO  bal-0
ELSE
 IF work-1  <  60
   ADD  amount-out  TO  bal-30
 ELSE
   IF work-1  <  90
     ADD  amount-out  TO  bal-60
   ELSE
     ADD  amount-out  TO  bal-90
END-IF
ADD  amount-out  TO  bal-t.
```

C++

```
FormattedNumber<8,2,1> work_1(0);
FormattedNumber<8,2,1> amount_out(0);
FormattedNumber<8,2,1> inv_amount(0);

FormattedNumber<8,2,1> bal_0(0);
FormattedNumber<8,2,1> bal_30(0);
FormattedNumber<8,2,1> bal_60(0);
FormattedNumber<8,2,1> bal_90(0);
FormattedNumber<8,2,1> bal_t(0);

FormattedNumber<1> oi_type();
FormattedNumber<7,2,1> oi_net();
FormattedNumber<7,2,1> oi_extra();
FormattedNumber<7,2,1> oi_carriage();
FormattedNumber<7,2,1> oi_discount();
FormattedNumber<7,2,1> oi_vat();
FormattedNumber<7,2,1> oi_e_vat();
FormattedNumber<7,2,1> oi_c_vat();
FormattedNumber<7,2,1> oi_paid();
FormattedNumber<3,2,1> oi_deduct_amt();
FormattedNumber<3,2,1> oi_deduct_vat();

void main() {
  inv_amount = oi_net + oi_extra +
    oi_carriage + oi_vat + oi_discount
    + oi_e_vat + oi_deduct_amt
    + oi_deduct_vat + oi_c_vat;

  if (oi_type == 3) {
    oi_paid *= -1;
  }
  if (oi_type == 2 || oi_type == 3) {
    amount_out = inv_amount - oi_paid;
  }

  if (work_1 < 30) {
    bal_0 += amount_out;
  } else {
    if (work_1 < 60) {
      bal_30 += amount_out;
    } else {
      if (work_1 < 90) {
        bal_60 += amount_out;
      } else {
        bal_90 += amount_out;
      }
    }
  }
  bal_t += amount_out;
}
```

Table 8.12: The tables shows a modified extract from the Applewood Accounting System (left), as part of a routine to handle invoices and account balances. The rightmost program is the C++ equivalent, translated from COBOL (left).

8.5.2 Intent Preservation

Per definition (see Definition 8.2.1), every language construct linked to a concept (under conditions) represents the same intent as all others linked to the same concept. Therefore, since our translation approach is based on concepts, the translation keeps the intent given that the concepts are translated correctly (see previous the sub-section about correctness). Nonetheless, it is interesting to demonstrate how preservation of intent works in some practical scenarios.

Consider the programs in Table 8.13, where the purpose is to find a specific Fibonacci number and then display that along with the next one. Compare this to the programs in Table 8.14, were the purpose is to find the greatest Fibonacci number under a given limit (100 in this case) and then display that, along with the difference between the Fibonacci number and the limit. In Table 8.13, we can see that the author intended to do a consecutive iteration to find the specific Fibonacci number. This is reflected in the translation to C++, where the consecutive iteration is represented as the most common way of expressing an iteration in C++, namely the `for` loop. In the programs in 8.14, on the other hand, the author did not intend a consecutive iteration since it is not known when to stop (and an index is not needed). Therefore, it is translated to a `while` loop representing the conditional loop concept.

Another example of translated intent is the last calculations in both Listing 8.13 and 8.14. In Listing 8.13, we can presume that the author had a reason for expressing `f = f + f1` as a full expression, rather than the shorthand version `ADD f1 TO f`. This intent is preserved in such a manner that the full expression is also present in the C++ code. Similarly, in Listing 8.14, the author used the shorthand notation `SUBTRACT f1 FROM lim`, instead of expressing the calculation as the full expression `COMPUTE lim = lim - f1`. This intent is also kept, since the calculation is expressed in short-hand notation in C++.

COBOL	C++

IDENTIFICATION DIVISION.
PROGRAM-ID. fibonacci-1.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 f1 PIC 9(12) VALUE 0.
77 f2 PIC 9(12) VALUE 1.
77 f PIC 9(12).
 	
PROCEDURE DIVISION.
PERFORM 8 TIMES
    ADD f1 TO f2 GIVING f
    MOVE f2 TO f1
    MOVE f TO f2
END-PERFORM
 	
DISPLAY f
COMPUTE f = f + f1
DISPLAY "Next: " f.
	FormattedNumber<12> f1(0);
	FormattedNumber<12> f2(1);
	FormattedNumber<12> f();
	FormattedNumber<18> i; // generated
	
	void main() {
	    for(i = 0; i < 8; i++) {
	        f = f1 + f2;
	        f1 = f2;
	        f2 = f;
	    }
	    cout << f << endl;
	    f = f + f1;
	    cout << "Next: " << f << endl;
	}

Table 8.13: The listing shows two equivalent programs for calculating the 9th and 10th Fibonacci number (34 and 55, respectively). The leftmost COBOL program is the original, and the rightmost C++ program has been translated from it. Inspired by the Rosetta Code implementation. [73]

COBOL	C++

IDENTIFICATION DIVISION.
PROGRAM-ID. fibonacci-2.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 f1 PIC 9(12) VALUE 0.
77 f2 PIC 9(12) VALUE 1.
77 f   PIC 9(12).
77 lim PIC 9(12) VALUE 100.
 	
PROCEDURE DIVISION.
PERFORM UNTIL f > lim
 ADD f1 TO f2 GIVING f
 MOVE f2 TO f1
 MOVE f TO f2
END-PERFORM
 	
DISPLAY f1
SUBTRACT f1 FROM lim
DISPLAY "Diff: " lim.
	FormattedNumber<12> f1(0);
	FormattedNumber<12> f2(1);
	FormattedNumber<12> f();
	FormattedNumber<12> lim(100);
	
	void main() {
	  while(f <= lim) {
	  f = f1 + f2;
	  f1 = f2;
	  f2 = f;
	  }
	  cout << f1 << endl;
	  lim -= f1;
	  cout << "Diff: " << lim << endl;
	}

Table 8.14: The listing shows two equivalent programs for calculating the greatest Fibonacci number under 100. The leftmost COBOL program is the original, and the rightmost C++ program has been translated from it. Inspired by the Rosetta Code implementation. [73]

8.5.3 Construct Preservation

Consider the COBOL program in Table 8.15. This piece of code only contains concepts which have exactly one concrete representation per language. Therefore, all unmodified constructs will remain intact during a back-and-forth translation as displayed in Table 8.16, where all unmodified constructs remain intact when more constructs are added.

COBOL	C++

IDENTIFICATION DIVISION.
PROGRAM-ID. example.
DATA DIVISION.
WORKING-STORAGE SECTION .
77 num COMP-1 VALUE 2.
 	
PROCEDURE DIVISION .
IF num LESS 3
  DISPLAY "Less than 3"
ELSE
  DISPLAY "Greater or equal to 3" .
	float num = 2f;
	
	void main() {
	  if (num < 3) {
	  cout << "Less than 3" << endl;
	  } else {
	  cout << "Greater or equal to 3"
	  << endl;
	  }
	}

Table 8.15: An example program, represented in COBOL (left) and C++ (right), where every concept only has one concrete representation.

COBOL	C++

IDENTIFICATION DIVISION.
PROGRAM-ID. example.
DATA DIVISION.
WORKING-STORAGE SECTION .
77 num COMP-1 VALUE 2.
 	
PROCEDURE DIVISION .
IF num LESS 3
  DISPLAY "Less than 3"
  DISPLAY "This is an added line"
  IF num LESS 2
  DISPLAY "Also less than 2"
  END-IF
ELSE
  DISPLAY "Greater or equal to 3" .
	float num = 2f;
	
	void main() {
	  if (num < 3) {
	  cout << "Less than 3" << endl;
	  cout << "This is an added line"
	  << endl;
	  if (num < 2) {
	  cout << "Also less than 2"
	  << endl;
	  }
	  } else {
	  cout << "Greater or equal to 3"
	  << endl;
	  }
	}

Table 8.16: A modified version of the programs in Table 8.15. A statement for printing `"This is an added line"` and the if-statement following that have been added. Notice that all unmodified language constructs in both languages are left intact when new constructs were added.

When back-and-forth translating code containing concepts which have more than one concrete representation (in our case: all short-hand operations, arithmetic expressions, and loops), language constructs might be changed, i.e. there is no guarantee that unmodified constructs will remain intact. For example, a piece of C++ code and its corresponding COBOL code are shown in Table 8.17. Translating that COBOL code back to C++ could result in the code presented in Listing 6, where `i += 1` and `i -= 1` have been replaced by `i++` and `i--`. Note that by our definition of a concept, this is a valid solution; however, it is not optimal since it is not the original representation. The reason for why this occurs is due to that the tool used, Medini QVT, does not support model synchronization. If another tool that supports model synchronization could have been used, such as Echo, the same QVT-R transformations could have been run and would have produced an optimal solution with the original constructs left intact.

C++	COBOL

`float num = 2f;`
`double i = 5.0;`
`void main() {`
`  if (num < 3) {`
`  i += 1;`
`  } else {`
`  i -= 1;`
`  }`
`  cout << num << endl;`
`}`
	`IDENTIFICATION DIVISION.`
	`PROGRAM-ID. example.`
	`DATA DIVISION .`
	`WORKING-STORAGE SECTION .`
	`77 num COMP-1 VALUE 2 .`
	`77 i COMP-2 VALUE 5 .`
	`PROCEDURE DIVISION .`
	`IF num LESS 3`
	`  ADD 1 TO i`
	`ELSE`
	`  SUBTRACT 1 FROM i`
	`END-IF`
	`DISPLAY num .`

Table 8.17: An example of a program, represented in C++ (left) and COBOL (right), which contains concepts that have several concrete representations; mainly, the short-hand operations in C++ can be expressed in 2 ways: `i++` or `i+=1`.

```
float num = 2f;
double i = 5.0;
void main() {
  if (num < 3) {
    i++;
  } else {
    i--;
  }
  cout << num << endl;
}
```

Listing 6: The equivalent of the C++ program presented in Table 8.17 after a back-and-forth transformation. Notice that the representation of the short-hand operations, increment and decrement, has gotten a different representation.

As evident by these examples, construct preservation is only optimal when translating concepts which have exactly one concrete representation per language. In other cases, construct preservation depends on which solution is chosen by Medini QVT and it seems to choose the solution whose relation was executed last, based on our experience.

8.6 Discussion

The described approach for implementing a source-to-source compiler has some notable advantages and disadvantages. First of all, the choice of having a model-driven approach was determined mainly by the transformation language. As motivated in Section 4.4, QVT-R was chosen due to it being a standard and its ability to handle

complex relations and conditions. The choice of the model-based approach leads to additional components due to the required text-to-model transformations, i.e. the Xtext parts. However, we did not deem many of the available text-based tools sufficient as alternatives. Another reason, far less trivial, is to show that a model-driven bidirectional source-to-source compiler for general-purpose languages actually works, since we have not seen any evidence of such an implementation in the literature.

The prominent alternative of the text-based transformation options considered by us was Boomerang. One advantage of using Boomerang would have been that transformations would have been well-formed, according to Boomerang's *lens* format. The QVT-R transformations specified by us have no such guarantee, as obvious by the back-and-forth translation examples in Section 8.5.3 (although this could, in theory, be guaranteed by for example Echo's least-change principle). The reasons why Boomerang ultimately was not chosen as the transformation language were mainly: 1) uncertainties about whether a C++ and COBOL grammar can be fully specified by Boomerang's syntax, and 2) uncertainties whether the expressiveness of Boomerang would allow us to specify complex concept-based transformations. Boomerang and other text-based options seem more suitable to deal with straightforward transformations. As such, we deem it possible that Boomerang may be a good fit for implementing a less complex source-to-source compiler, with non-concept based translation.

As explained in the Method Section 4.4, using Medini QVT was not our first choice, but rather a back-up option as Echo did not work. As a consequence of Medini QVT not supporting model synchronization, the prototype does not adhere to RQ3c. However, since the transformations are implemented in QVT-R, another tool, such as the one being developed by the Eclipse Foundation [35], could, in theory, be used to run the transformations; although this is a cause for concern regarding validity. We do not know whether such a tool will be available; nor do we know how well our transformations would work in such a tool. For example, there is a philosophical question of what exactly Echo's least-change principle means that needs to be at least partially answered before the approach becomes of practical interest. Echo's default change measure of graph editing distance might be employed as a naive method of determining least change in the language models, but what really is measured then is the grammar-wise differences. However, we estimate, based on very small examples run in Echo that using a tool implementing model synchronization will not be a major problem, as long as the tool is efficient and follows the QVT-R standard. The problem rather lies in producing a tool capable of synchronizing large models. If no such tool will be produced, other approaches might have to be considered.

There are several concerns regarding a scale-up of our prototype. One such concern is assuring validity. Recall that we differentiate between transformation validity and specification validity (see Section 4.4.6). Transformation validity is largely guaranteed by using a bidirectional transformation language which guarantees validity by construct; together with a valid metamodel and a valid grammar, validity by construct can guarantee that RQ3a is answered (and part of RQ3b as well). Specification validity, on the other hand, is not as easy to guarantee. Since our prototype is of small scale, and due to time limitations, we were able to man-

ually verify specification validity by testing a few representative examples. These examples only contain language constructs supported by our prototype, meaning that they only express very limited functionality. It could be argued that these examples are not comprehensive enough to guarantee specification validity. However, it is outside the scope of this work to present full test coverage. Instead, the main purpose of the examples is to illustrate the feasibility of our approach, which we do on a small scale by providing code from an industry project (Applewood Computing Accounting System). Since most of the pieces of code in the Applewood Accounting System that can be supported by our prototype are quite similar in nature, we did not deem it useful to include several examples.

If our approach should be used to implement a complete translator, for e.g. COBOL and C++, a more comprehensive testing approach is required. What this would entail is unclear, as we are currently not aware of any standardized methods of testing QVT-R code. Since QVT-R acts on relations which specify properties that must hold, property-based testing, used in e.g. functional programming languages, is certainly one possible alternative. Another area of interest for testing is the unidirectional conditions applied in some relations. Furthermore, the Xtext grammars used should also be tested. Even though the grammars may have been tested in their original format, many changes will have been made once converted to Xtext grammars. For example, parts of our Xtetx grammars have been removed or refactored, and we are aware that they contain some errors in more complex statements. Testing Xtext grammars should be more straightforward though, as Xtext has built-in support for this.

Another potential validity threat regarding a scale-up of our prototype is handling advanced language constructs present in one language but not the other(s). We dealt with one such construct, COBOL's special data types, by creating a library which contains functionality for emulating COBOL functionality in C++. By nature, COBOL does have its peculiar language features, but in general it is not such a complex language; C++, on the other hand, is. Representing special C++ features in COBOL might prove difficult. Take, for example, the notion of scopes in C++, which is not by itself even a complex language feature. There are essentially two ways of approaching a translation of scopes when also keeping the original structure after a back-and-forth translation. Either they are emulated in COBOL, e.g. by inserting comments as illustrated in Table 8.18 (information is kept), or scopes are flattened when translating from COBOL to C++ and are arbitrarily generated when translating from C++ to COBOL. The first solution is by far the most efficient one, however, it introduces concepts which have no meaning in the COBOL code. Since one purpose of a bidirectional source-to-source compiler is to allow simultaneous development in both languages, this is not ideal. The second approach does not have this problem; however, it may require too much computing power when synchronizing models, depending on the tool's underlying implementation. For example, the very small piece of COBOL code in Listing 8.19 can be translated in several ways to C++, 3 of which are shown. Having many statements like these in Echo would result in huge SAT problems. It seems like there is no trivial optimal solution for cases such as these and we might have to accept the fact that everything cannot be optimally translated due to language incompatibility.

COBOL	C++

...
* SCOPE-START
  ADD X TO Y.
  MUTLIPLY Y BY X.
* SCOPE-FINISH
...
	...
	{
	  y += x;
	  x *= y;
	}
	...

Table 8.18: An example of retaining information about scopes in COBOL when translating from C++. Scope are marked in COBOL as comments.

COBOL	C++
... ADD X TO Y. MUTLIPLY Y BY X. ...	... y += x; { x *= y; } ...

	...
	y += x;
	x *= y;
	...

	...
	{
	  y += x;
	  x *= y;
	}
	...

Table 8.19: An example of the problem faced when not retaining any scope information in COBOL. The examples shows that there are multiple valid ways (right side) of translating the COBOL code (left side) to C++ code, each resulting in a different scope declaration.

In Section 8.4.3 we presented a problem which entailed that if our approach to enforce order in collections was used, it was only possible to transform statements that had corresponding one-to-one mappings in terms of the amount of elements produced; that is, statements which need to be represented by two or more statements in the other language cannot be transformed while also keeping order. It is worth mentioning that we think this problem is solvable, but due to time limitation and limited knowledge of QVT-R, we did not manage to solve it. One possible solution could be to represent all collections as linked lists. As such, the order could be kept by treating collection elements at an atomic level, with the ability to look forwards and backwards in the list to determine which statements should be merged or split.

Finally, we discuss the use of concepts. The reason for defining and using concepts is to maintain intent in translated code, as per RQ3d. Defining concepts, and thereby also intent, is a somewhat philosophical and personal process, involving

a human element. The concept-centric transformation mechanism is only a means to maintain the intent that is defined. That is, intent preservation is fully determined by how well concepts are defined. Therefore, if concepts are not defined with care, this mechanism may not improve the quality of translation.

To conclude, we acknowledge that there are several known aspects which need further examination in order to determine if our approach is viable on a larger scale. As such, the work presented in this thesis should be considered as an initial step and a roadmap for further work in bidirectional model-driven language translation.

9
Conclusion

In this thesis, we have presented an approach for building a bidirectional model-driven source-to-source compiler, which translates between COBOL and C++ code. Our approach extends the work of Yellin [55] in a number of aspects. First, our approach is model-driven. This opens up for opportunities in the modelware technical space, such as extracting diagrams, graphical coding, and usage of other tools based on the de-facto standard Ecore. Furthermore, having a model-driven approach is advantageous when implementing transformations, due to the use of inheritance and object-oriented features, which enables re-use of code. Second, our approach uses state-of-the-art bidirectional transformation techniques, based on the established standard QVT-R. This means that there are, and will be, several tools able to run the transformations, including support for handling non-bijective relations and debugging. Furthermore, these transformation techniques allow us to express complex transformation patterns. Third, our approach of defining an intermediate model is based on concepts intended to preserve intent in code. Whereas Yellin [55] only talks about the consequences of choosing GCD on different abstraction levels, we define an approach that is consequence-aware and has a clear focus to keep intent in code when translated. Fourth, we are able to express complex context-aware relations. Whereas previous attempts mostly focused on translation of individual language constructs, we are able to translate language constructs based on their relation to other language constructs present in the code. As such, several language constructs can be related to the same concept under different conditions. Finally, we consider if the abstraction level of the GDC between language constructs can be raised by emulating features of one language in the other.

Our prototype shows that a model-driven translation approach is indeed feasible. We have shown that RQ3a and RQ3b can be answered using model-based grammars, valid metamodels, bidirectional transformation languages (thus gaining transformation validity by construct), and a standard library emulating language features. We have also shown that intent can be defined and preserved when code is translated by the use of concepts which inherently represent intent, thus answering RQ3d. RQ1 helped us focus on which language constructs were important to support, resulting in that basic common constructs, such as if, loop, arithmetic, and print statements were implemented in the developed prototype. RQ2 helped us choose the target language C++ as a suitable option for translation, due to its expressiveness, and similarities with COBOL.

9.1 Limitations

The developed prototype is only intended as a proof of concept and therefore has serious limitations in its completeness (only subsets of COBOL and C++ are supported). The language constructs supported in our prototype are of simple nature; therefore, it was easy to verify specification validity and answer RQ3b. All COBOL data types were in fact emulated, and both a complete COBOL and a C grammar, with a few exceptions, were specified. The supported language constructs are, instead, limited by specified transformations.

We have not shown that our approach is feasible for all kinds of scenarios that might occur when translating an entire language. Nor have we shown generality in translating any two languages. Although we believe our approach to be fairly generalizable, we also believe the success hinges largely on four factors:

- whether an Xtext grammar can be created for any language part of the translation,
- if the languages share relatively similar constructs such that sound concepts can be defined,
- if the translation involves any construct which is represented as one statement in one language and multiple statements in the other, the problem described in Section 8.4.3 must be solved, and
- whether a suitable tool, which supports model synchronization, can be used.

As illustrated in Section 8.5, the drawback of using Medini QVT is that it does not support model synchronization, meaning that a back-and-forth translation could potentially result in different pieces of code, even though no changes were made. As such, our prototype does not satisfy RQ3c. If, however, a QVT-R tool which implements model synchronization (e.g. a working version of Echo) is available, the translations devised here should theoretically work with that tool as well. Therefore, we theoretically see no obstacles in our general approach in regards to RQ3c.

9.2 Future Work

This final section closes the thesis with our thoughts about future opportunities in the different areas related to each result sections. Overall, the foundations of each result can be seen as successes, but the implementation can improve and mature out of the prototype-stadium.

9.2.1 On the Language Construct Analysis

A natural extension of the work already performed in the language analysis is to analyze more systems, and compare systems between different areas (open source, closed source, financial, automotive etc.). When applying the method to new systems, the analyzer might need to be changed. For example, some systems might

need more pre-processing before being compilable with GnuCOBOL. Some systems might not be fully compilable, but still analyzable, with some changes to the tool that suppresses compiler-related errors.

9.2.2 On Emulating COBOL Data Types

Directly, the data types emulated can be improved by evolving the software behind: more features can be supported (more overloads), and a proper test suite can be added. The code can be further optimized for readability, or for execution speed. Documentation can be written. Beyond emulation, the transformations into the emulated types can be improved in order to make use of these features. An open question is still how to handle overflow signal handling: exception, global flag, pre-test of whether an operation will overflow, or something else?

9.2.3 On the Source-to-Source Compiler

In Section 8.6, we mentioned that validity and complex language constructs are two major aspects which need further attention in order to evaluate our approach to build a source-to-source compiler on a larger scale than the prototype presented in this theses. Below, we present four additional areas, not included in our scope, where our work can be further evolved.

9.2.3.1 Choosing a Good Solution

Our idea of concepts is intended to capture intent in code. Oftentimes, a concept has several different concrete representations in each programming language. The question of how to choose a good solution, among these different representations, then comes to mind. In the case when something is modified and is to be back-translated, the answer to this question is easy: choose the solution which entails the least change. However, if a new concept, which has several concrete representations, is added, the question becomes harder. It could be argued that any solution will suffice since a concept should reflect the same intent, no matter the concrete representation. However, oftentimes it would seem preferable that the same coding style is used and that the choice is consistent. One way to achieve this could be to implement a configurable user profile that will determine the choice in different scenarios. Another way could be to implement a stochastic mechanism that, based on the nearby code, measures and picks the option with the highest probability.

9.2.3.2 Eliminating the Intermediate Model

The motivation for using an intermediate model in our transformation chain is that it eliminates the number of transformations written for concepts which have multiple concrete representations. Furthermore, it allows extension to more languages, without jeopardizing existing transformations. With that said, QVT-R allows transformations between multiple models. Therefore, if dealing with situations where concepts tend to have few concrete representations, having an intermediate model

might become superfluous and eliminating the intermediate model might be a feasible alternative.

Listing 9.1: A QVT-R relation which is able to transform if-statements between 4 different languages

```
relation ifStatementRelation {
  enforce domain cob cobolIf: cob::IfStatement {
    condition = cobolCond : cob::Condition{},
    ifStatements = cobIfStat : cob::StatementList{},
    elseStatements = cobElseStat : cob::StatementList{}
  };
  enforce domain c cIf: c::IfStatement {
    condition = cCond : c::Condition{},
    ifStatements = cIfStat : c::Statement{},
    elseStatements = cElseStat : c::Statement{}
  };
  enforce domain j javaIf: java::IfStatement {
    condition = javaCond : java::Condition{},
    ifStatements = javaIfStat : java::Statement{},
    elseStatements = javaElseStat : java::Statement{}
  };
  enforce domain py pyIf: python::IfStatement {
    condition = oythonCond : python::Condition{},
    ifStatements = pyIfStat : python::Statement{},
    elseStatements = pyElseStat : python::Statement{}
  };
  where {
    conditionRelation(cobolCond, cCond, javaCond, pyCond);
    statementRelation(cobIfStat, cIfStat, javaIfStat, pyIfStat);
    statementRelation(cobElseStat, cElseStat, javaElseStat, pyElseStat);
  }
}
```

Consider the relation displayed in Listing 9.1, which can transform if-statements between COBOL, C++, Java, and Python. In a situation such as the one in Listing 9.1, where there is a clear one-to-one relation between corresponding statements, having one relation is more convenient than having one relation for each language to an intermediate model. However, as soon as there are multiple representations, or if one statement is represented by multiple corresponding statements in another language, this approach seems to become cumbersome. However, there might be a feasible way to express such scenarios in QVT-R, but we have discovered no such way.

9.2.3.3 Translation within a Language

The techniques presented can also be used to translate code within a language. Instead of having the COBOL and C++ stages on each side of the intermediary model, both sides could have the same language attached (e.g. two COBOL models). The result would be that language constructs could be translated within COBOL, as long as they satisfy the same concept. This could for example be used to refactor code to comply with a certain coding standard, given that one concept representation (in code) is marked as preferred over the others.

9.2.3.4 Usages beyond Translation

The parsing and identification of concepts may also be used to other ends than translation, or translation to other models than text-based ones. For example, the identified concepts may be used to generate documentation of code or abbreviated excerpts of it. It could also be used for pre-processing as input into machine learning or search applications. It could, for example, be used to extend the method presented by Bitbucket for code search, where the concept of definitions (variables, classes, functions etc.) is identified and shown higher up in search results [74].

Bibliography

[1] Micro Focus. Academia needs more support to tackle the it skills gap. https://www.microfocus.com/about/press-room/article/2013/academia-needs-more-support-to-tackle-the-it-skills-gap/, 2013. (Accessed on 04/03/2017).

[2] Gary Anthes. Cobol coders: Going, going, gone? http://www.computerworld.com/article/2554071/it-careers/cobol-coders--going--going--gone-.html, 2006. (Accessed on 04/03/2017).

[3] Tiobe. Tiobe index for january 2017. http://www.tiobe.com/tiobe-index/, 01 2017. (Accessed on 01/16/2017).

[4] Mark Driver. Introducing the gartner programming language index for 2014. http://blogs.gartner.com/mark_driver/2014/10/02/gartner-programming-language-index-for-2014/, October 2014. (Accessed on 01/16/2017).

[5] Datamonitor. Cobol – continuing to drive value in the 21st century. https://dl.microfocus.com/000/COBOL_continuing_to_drive_value_in_the_21st_Century_tcm21-23652.pdf, November 2008. (Accessed on 01/17/2017).

[6] GnuCobol. Gnucobol. https://sourceforge.net/projects/open-cobol/.

[7] Heirloom paas. http://heirloomcomputing.com/. (Accessed on 01/17/2017).

[8] Toshio Suganuma, Toshiaki Yasue, Tamiya Onodera, and Toshio Nakatani. Performance pitfalls in large-scale java applications translated from cobol. In *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA Companion '08, pages 685–696, New York, NY, USA, 2008. ACM. URL: http://doi.acm.org/10.1145/1449814.1449822, doi:10.1145/1449814.1449822.

[9] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory (Volume: 2, Issue: 3, September 1956)*, 1955. URL: https://chomsky.info/wp-content/uploads/195609-.pdf.

[10] John W Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. *Proceedings of the International Comference on Information Processing, 1959*, 1959. URL: http://www.softwarepreservation.org/projects/ALGOL/paper/Backus-Syntax_and_Semantics_of_Proposed_IAL.pdf.

[11] ISO/IEC. Iso/iec 14977:1996(e) - information technology - syntactic metalanguage - extended bnf. http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf, 1996. (Accessed on 10/05/2017).

[12] Donald E. Knuth. On the translation of languages from left to right. *Information and Control (Volume 8, Issue 6, December 1965, Pages 607-639*, 1965. URL: http://www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/knuth65.pdf.

[13] Seppo Sippu; E. Soisalon-Soininen. *Parsing theory volume 2: LR(K) and LL(K) parsing*. Springer-Verlag New York, 1990.

[14] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. *Springer*, 1978. URL: http://bat8.inria.fr/~lang/papers/icalp74/icalp74.pdf.

[15] Bison. Gnu bison. https://www.gnu.org/software/bison/.

[16] Terence Parr and Kathleen Fisher. Ll(*): The foundation of the antlr parser generator. *SIGPLAN Not.*, 46(6):425–436, June 2011. URL: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1993316.1993548, doi:10.1145/1993316.1993548.

[17] Jeffrey D. Ullman Alfred V. Aho, Ravi Sethi and Monica S. Lam. *Compilers: Principles, Techniques, and Tools, Second Edition ("Purple dragon book")*. Springer, 2006.

[18] E. Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, Sept 2003. doi:10.1109/MS.2003.1231147.

[19] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41, Sept 2003. doi:10.1109/MS.2003.1231149.

[20] Object Management Group. Meta object facility (mof) 2.5 core specification. Technical report, Object Management Group, 2015. URL: http://www.omg.org/spec/MOF/2.5/.

[21] Perdita Stevens. A landscape of bidirectional model transformations. In *Generative and transformational techniques in software engineering II*, pages 408–424. Springer, 2008.

[22] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[23] Christopher M. Poskitt, Mike Dodds, Richard F. Paige, and Arend Rensink. Towards rigorously faking bidirectional model transformations. In J. Dingel, J. De Lara, L. Lúcio, and H. Vangheluwe, editors, *Workshop on Analysis of Model Transformations, AMT 2014*, volume 1277 of *CEUR-WS*, pages 70–75, Aachen, September 2014. RWTH Aachen, Germany. URL: http://doc.utwente.nl/93308/.

[24] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. *Bidirectional Transformations: A Cross-Discipline Perspective*, pages 260–283. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. URL: http://dx.doi.org/10.1007/978-3-642-02408-5_19, doi:10.1007/978-3-642-02408-5_19.

[25] Andy Schürr. *Specification of graph translators with triple graph grammars*, pages 151–163. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. URL: http://dx.doi.org/10.1007/3-540-59071-4_45, doi:10.1007/3-540-59071-4_45.

[26] Anthony Anjorin, Marius Lauder, Sven Patzina, and Andy Schürr. emoflon: Leveraging emf and professional case tools. In *Tagungsband der INFORMATIK 2011 Lecture Notes in Informatics*, volume 192. TUbiblio, July 2011. URL: http://tubiblio.ulb.tu-darmstadt.de/72868/.

[27] Mote – tgg-based model transformation engine | mdelab. https://www.hpi.uni-potsdam.de/giese/public/mdelab/mdelab-projects/mote-a-tgg-based-model-transformation-engine/. (Accessed on 05/12/2017).

[28] Tgg interpreter. http://www-old.cs.uni-paderborn.de/en/research-group/software-engineering/research/projects/tgg-interpreter.html. (Accessed on 05/12/2017).

[29] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A comparison of incremental triple graph grammar tools. *Electronic Communications of the EASST*, 67, 2014.

[30] Object Management Group. Meta object facility (mof) 2.0 query/view/transformation specification. Technical report, Object Management Group, 2016. URL: http://www.omg.org/cgi-bin/doc?formal/2016-06-03.

[31] Perdita Stevens. Bidirectional model transformations in qvt: semantic issues and open questions. *Software & Systems Modeling*, 9(1):7, 2008. URL: http://dx.doi.org/10.1007/s10270-008-0109-9, doi:10.1007/s10270-008-0109-9.

[32] Medini qvt. http://projects.ikv.de/qvt. (Accessed on 05/29/2017).

[33] Nuno Macedo, Tiago Guimaraes, and Alcino Cunha. Model repair and transformation with echo. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 694–697. IEEE, 2013.

[34] Alloy. http://alloy.mit.edu/alloy/index.html. (Accessed on 05/13/2017).

[35] Eclipse qvtd (qvt declarative). https://projects.eclipse.org/projects/modeling.mmt.qvtd. (Accessed on 05/13/2017).

[36] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. *JTL: A Bidirectional and Change Propagating Transformation Language*, pages 183–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. URL: http://dx.doi.org/10.1007/978-3-642-19440-5_11, doi:10.1007/978-3-642-19440-5_11.

[37] Shinya Kawanaka and Haruo Hosoya. bixid: A bidirectional transformation language for xml. *SIGPLAN Not.*, 41(9):201–214, September 2006. URL: http://doi.acm.org/10.1145/1160074.1159830, doi:10.1145/1160074.1159830.

[38] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. *Dual Syntax for XML Languages*, pages 27–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. URL: http://dx.doi.org/10.1007/11601524_2, doi:10.1007/11601524_2.

[39] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. *SIGPLAN Not.*, 43(1):407–419, January 2008. URL: http://doi.acm.org/10.1145/1328897.1328487, doi:10.1145/1328897.1328487.

[40] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological spaces: An initial appraisal. In *International Symposium on Distributed Objects and Applications, DOA 2002*, 2002. URL: http://doc.utwente.nl/55814/.

[41] M. Alanen and I. Porres. *A Relation Between Context-free Grammars and Meta Object Facility Metamodels*. TUCS technological report. Turku Centre for Computer Science, 2004. URL: https://books.google.se/books?id=prQTAwAACAAJ.

[42] Manuel Wimmer and Gerhard Kramler. *Bridging Grammarware and Modelware*, pages 159–168. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. URL: http://dx.doi.org/10.1007/11663430_17, doi:10.1007/11663430_17.

[43] Andreas Kunert. Semi-automatic generation of metamodels and models from grammars and programs. *Electron. Notes Theor. Comput. Sci.*, 211:111–119, April 2008. URL: http://dx.doi.org/10.1016/j.entcs.2008.04.034, doi:10.1016/j.entcs.2008.04.034.

[44] Sven Efftinge and Markus Völter. oaw xtext: A framework for textual dsls. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, page 118, 2006.

[45] Terence Parr. Antlr parser generator. http://www.antlr3.org/. (Accessed on 05/09/2017).

[46] Terence Parr and Kathleen Fisher. Ll(*): The foundation of the antlr parser generator. *SIGPLAN Not.*, 46(6):425–436, June 2011. URL: http://doi.acm.org/10.1145/1993316.1993548, doi:10.1145/1993316.1993548.

[47] Jianan Yue. Transition from ebnf to xtext. In *PSRC@MoDELs*, 2014.

[48] Alexander Bergmayr and Manuel Wimmer. Generating metamodels from grammars by chaining translational and by-example techniques. In *MDEBE@ MoDELS*, pages 22–31, 2013.

[49] Penny Barbe. Techniques for automatic program translation. In JULIUS T. TOU, editor, *Software Engineering*, pages 151 – 165. Academic Press, 1970. URL: http://www.sciencedirect.com/science/article/pii/B9780123954954500192, doi:https://doi.org/10.1016/B978-0-12-395495-4.50019-2.

[50] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. The problem of programming communication with changing machines: A proposed solution. *Commun. ACM*, 1(8):12–18, August 1958. URL: http://doi.acm.org/10.1145/368892.368915, doi:10.1145/368892.368915.

[51] LLVM. The llvm compiler infrastructure. http://llvm.org/. (Accessed on 05/29/2017).

[52] Paul F. Albrecht, Phillip E. Garrison, Susan L. Graham, Robert H. Hyerle, Patricia Ip, and Bernd Krieg Brückner. Source-to-source translation: Ada to pascal and pascal to ada. In *Proceedings of the ACM-SIGPLAN Symposium on The ADA Programming Language*, SIGPLAN '80, pages 183–193, New York, NY, USA, 1980. ACM. URL: http://doi.acm.org/10.1145/800004.807949, doi:10.1145/800004.807949.

[53] Bernd Krieg-Brückner. *Language Comparison and Source-to-Source Translation*, pages 299–304. Springer Berlin Heidelberg, Berlin, Heidelberg, 1984. URL: http://dx.doi.org/10.1007/978-3-642-46490-4_26, doi:10.1007/978-3-642-46490-4_26.

[54] Jeannette M. Wing. Computational thinking. *Commun. ACM*, 49(3):33–35, March 2006. URL: http://doi.acm.org/10.1145/1118178.1118215, doi:10.1145/1118178.1118215.

[55] Daniel M. Yellin. *Attribute Grammar Inversion and Source-to-source Translation (Compilers)*. PhD thesis, Columbia University, New York, NY, USA, 1987. AAI8724115.

[56] Donald E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968. URL: http://dx.doi.org/10.1007/BF01692511, doi:10.1007/BF01692511.

[57] D. M. Yellin and E. M. M. Mueckstein. The automatic inversion of attribute grammars. *IEEE Transactions on Software Engineering*, SE-12(5):590–599, May 1986. doi:10.1109/TSE.1986.6312955.

[58] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Q.*, 28(1):75–105, March 2004. URL: http://dl.acm.org/citation.cfm?id=2017212.2017217.

[59] Applewood Computing. Applewood computing accounting system. https://sourceforge.net/projects/acas/. (Accessed on 01/20/2017).

[60] Vs cobol ii grammar version 1.0.4. http://www.cs.vu.nl/grammarware/vs-cobol-ii/. (Accessed on 05/24/2017).

[61] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[62] Grammar list. http://www.antlr3.org/grammar/list.html. (Accessed on 05/29/2017).

[63] Oracle. Java se: Java language and virtual machine specifications. https://docs.oracle.com/javase/specs/. (Accessed on 10/05/2017).

[64] ISO/IEC. Iso/iec 9899:1999 - programming languages - c. https://www.iso.org/standard/29237.html, 1999. (Accessed on 10/05/2017).

[65] ISO/IEC. Iso/iec 14882:2011 - information technology - programming languages - c++. https://www.iso.org/standard/50372.html, 2011. (Accessed on 10/05/2017).

[66] Microsoft. C# language specification. https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/language-specification. (Accessed on 10/05/2017).

[67] IBM. Ibm cobol language reference, fifth edition. http://math.uni.lodz.pl/~arogow/os390/podr/cobol-manual.pdf, 1998. (Accessed on 10/05/2017).

[68] IBM. Ibm knowledge center: Heap. https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.2.0/com.ibm.zos.v2r2.ceea500/clheap.htm. (Accessed on 09/05/2017).

[69] Micro Focus. Micro focus online documentation: Display. http://documentation.microfocus.com/help/index.jsp?topic=%2FGUID-0E0191D8-C39A-44D1-BA4C-D67107BAF784%2FHRLHLHPDF80D.html.

[70] Eli Bendersky. Variadic templates in c++. http://eli.thegreenplace.net/2014/variadic-templates-in-c/. (Accessed on 05/12/2017).

[71] Chapter 4 - the class file format. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7. (Accessed on 05/28/2017).

[72] Ocl 2.0. http://www.omg.org/spec/OCL/2.0/. (Accessed on 05/29/2017).

[73] Rosetta code: Fibonacci sequence in cobol. http://rosettacode.org/wiki/Fibonacci_sequence#COBOL. (Accessed on 05/29/2017).

[74] Bitbucket. Introducing code aware search for bitbucket cloud. https://blog.bitbucket.org/2017/05/02/introducing-code-aware-search-for-bitbucket-cloud/. (Accessed on 05/26/2017).