Innovations in Systems and Software Engineering
https://doi.org/10.1007/s11334-024-00568-z

SPECIAL ISSUE ON MODEL-DRIVEN ENGINEERING AND SYSTEM ANALYSIS AND MODELLING

Agile model-driven re-engineering

Kevin Lano[1] · Howard Haughton[2] · Ziwen Yuan[1] · Hessa Alfraihi[3]

Received: 2 January 2024 / Accepted: 27 May 2024

Abstract
In this paper we describe an Agile model-driven engineering (MDE) approach, AMDRE, for the re-engineering of legacy systems. The objective is to support the reuse of business-critical functionality from such systems and the porting of legacy code to modernised platforms, together with technical debt reduction to improve the system maintainability and extend its useful life. AMDRE uses a lightweight MDE process which involves the automated abstraction of software systems to UML specifications and the interactive application of refactoring and rearchitecting transformations to remove quality flaws and architectural flaws. We demonstrate the approach on Visual Basic, COBOL and Python legacy codes, including a finance industry case. Significant quality improvements are achieved, and translation accuracy over 80% is demonstrated. In comparison to other MDE re-engineering approaches, AMDRE does not require high MDE skills and should be usable by mainstream software practitioners.

Keywords Program abstraction · Model-driven engineering · Refactoring · Re-engineering

1 Introduction

Legacy code systems can cause significant business costs to organisations which depend upon these often poorly-maintained and poorly-documented software assets [34, 54, 68]. Apart from the direct maintenance costs of legacy systems, dependence on these systems can block business growth and evolution [16, 67], reduce developer morale and reduce the resources available for business innovation.[1] There can also be negative consequences for wider society. For example, during the COVID pandemic, legacy software issues caused delays to economic response measures in the US [2]. Legacy code platforms are typically less energy efficient than modern platforms,[2] leading to poor environmental sustainability. Flaws in legacy code can also result in wasted energy resources, due to inefficient coding practices, or increased binary size, due to dead code [16, 54, 67]. Even for relatively recent machine learning (ML) systems, a range of design flaws may be present, reducing performance and causing significant maintenance costs and excessive energy use in operation [63]. The high carbon footprint of ML systems means that improvements to their energy efficiency could produce significant reductions in CO_2 emissions [51].

Thus an effective re-engineering approach to support software asset recovery and modernisation could be a key enabler for businesses. There have been three main obstacles to the use of re-engineering in practice:

1. The manual effort required for re-engineering processes
2. The need to assure semantic preservation of the original code functionality by the new system

[1] https://vfunction.com/blog/how-much-does-it-cost-to-maintain-legacy-software-systems/.

[2] https://www.mavensolutions.tech/blog/cost-of-legacy-systems/.

✉ Kevin Lano
kevin.lano@kcl.ac.uk

Howard Haughton
howard.haughton@gmail.com

Ziwen Yuan
ziwen.yuan@outlook.com

Hessa Alfraihi
haalfraihi@pnu.edu.sa

1 Department of Informatics, King's College London, London, UK

2 Holistic Risk Solutions Ltd, Croydon, UK

3 Information Systems Department, Computer and Information Sciences, Princess Nourah Bint Abdulrahman University, Riyadh, Saudi Arabia

Published online: 18 June 2024

3. The high degree of variability in re-engineering tasks, due to the large number of different programming languages and environments in use.

To address these challenges we propose: (i) increased automation via the use of model-driven engineering (MDE) tools; (ii) verification of re-engineering steps in certain cases (such as numerical computations) and model-based testing (MBT) for automated testing in other cases; (iii) increased flexibility via the use of agile methods, and the use of lightweight tools which can be directly customised by end-users to rapidly construct new re-engineering solutions.

From the mid 1980s–early 1990s, various model and specification-based approaches were proposed for reverse-engineering legacy systems, to create precise documentation from code, and to restructure and improve system quality, as a means of prolonging the life of a system or as a step towards the production of a modernised system in a new programming language or platform [7, 47, 66]. These concepts of model-driven modernisation (MDM) were formalised by the Object Management Group (OMG) in their Architecture-driven modernisation (ADM) framework [60]. The envisaged MDM/ADM however uses heavyweight and strictly staged MDE processes, involving multiple models at multiple levels of abstraction, elaborate metamodels, and multiple model transformations [15]. Since the publication of the Agile manifesto [4] in 2001, the software industry has been moving to a more agile development approach for new system construction in many software application areas, and consequently there is also increasing interest in more agile and lightweight approaches to re-engineering [35, 68]. Use of ADM also requires significant expertise in MDE techniques, and there are limited numbers of software practitioners with such skills.

Based on our experience with the recovery and modernisation of legacy applications [38, 41], we evolved an agile MDE re-engineering approach (AMDRE), using iterations of steps 1 to 4 of Fig. 1 on parts of a source system. This approach uses grammar-based techniques requiring only a basic level of MDE knowledge.

Step 1 (Sect. 3) uses a combination of the ANTLR parser engine[3] and the Concrete Syntax Transformation Language ($\mathcal{CSTL}$) of [40] to represent system abstractions in UML and the Object Constraint Language (OCL) [55]. We use UML/OCL to represent system abstractions because these languages are widely-used in the software industry, are international standards, and are supported by large numbers of tools both for analysis and code generation/forward engineering. Thus producing system models in UML/OCL enables organisations to have confidence that these models will have lasting value and will not themselves become unusable legacy artifacts.

In step 2, re-engineering specialists perform refactoring operations on the abstracted specification, to remove or reduce quality flaws (Sect. 4). The system architecture can also be restructured in this step in order to satisfy clean architecture principles [49] such as the Interface Segregation Principle (ISP) and Acyclic Dependencies Principle (ADP) (Sect. 4.3).

Step 3 involves input from system experts to link system requirements to the abstracted system model, with the assistance of program summarisation tools. A high-level requirements specification can be produced (Sect. 6).

Step 4 generates an executable version of the re-engineered system for a new platform, together with a test suite for the system (Sect. 7). For forward engineering we use the AgileUML MDE toolset [18].

Because of the high variability of legacy re-engineering tasks and projects, close customer liaison is essential as part of this agile process, in order to receive rapid feedback and build understanding of the specific legacy system context across the re-engineering team [68]. In particular, steps 2 and 3 are interactive and iterative, whilst steps 1 and 4 can be entirely automated.

Automation of process steps helps to accelerate re-engineering, and is important to ensure the consistent correctness of re-engineering processes, that is, to ensure that the semantics of the source application is accurately translated to the target. An initial 'sprint 0' iteration will usually be performed to assess the requirements of the re-engineering project, and to prioritise specific re-engineering actions. This results in a 'project backlog' of re-engineering tasks, subsets of which will be handled as the 'iteration backlogs' of specific AMDRE iterations. The highest priority tasks remaining in the product backlog are generally selected for the iteration backlog of the next iteration.

The research questions are:

- $RQ1$: Can effective and reliable re-engineering processes be supported using ANTLR and $\mathcal{CSTL}$?
- $RQ2$: Does the AMDRE approach significantly reduce the manual workload of re-engineering tasks?

The overall aim is to reduce the human effort and resources needed for re-engineering and to enable businesses to incorporate rapid re-engineering and reuse into agile development processes.

Section 2 looks at the specific challenges of BASIC, COBOL and Python re-engineering, and Sect. 3 details how we address these challenges using semantic abstraction of source code. Section 4 describes how abstracted source code can be analysed and improved. Section 5 addresses the issue of semantic preservation, Sect. 6 considers the extraction of specifications, and Sect. 7 describes target code and test case synthesis. Section 8 gives a detailed evaluation, Sect. 9 con-

[3] https://antlr.org.

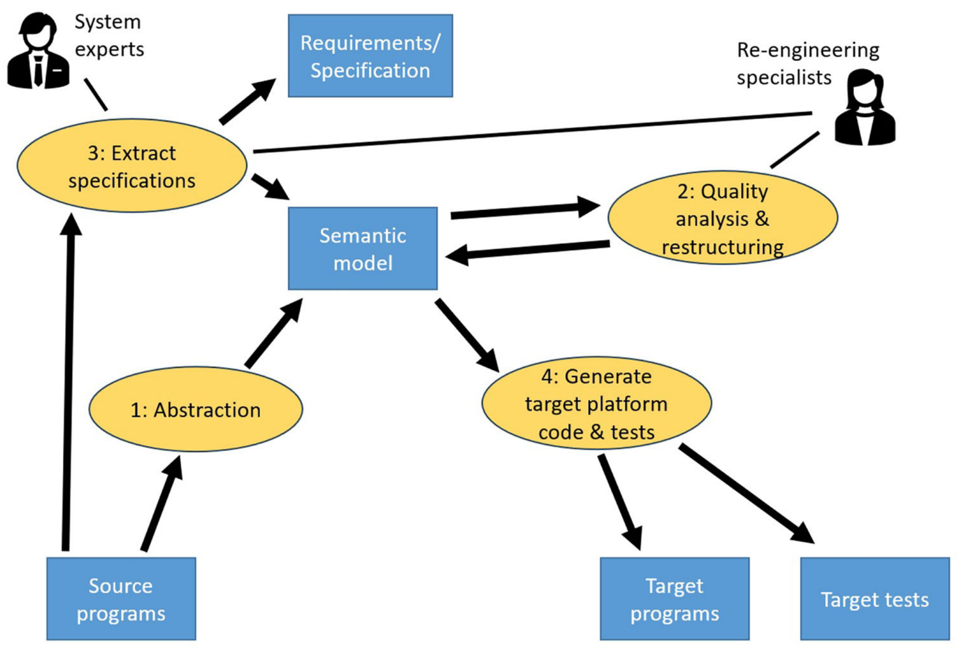
System experts
Requirements/ Specification
Re-engineering specialists
3: Extract specifications
2: Quality analysis & restructuring
Semantic model
1: Abstraction
4: Generate target platform code & tests
Source programs
Target programs
Target tests

Fig. 1 AMDRE iteration process steps

siders threats to validity, and Sect. 10 gives a comparison with related work. Finally, Sect. 11 gives conclusions and potential future work.

2 Challenges of legacy code abstraction

Legacy code can be of recent vintage [5, 62], including Python-based machine learning systems [63], however code in old languages such as versions of COBOL [61], BASIC [36] or PL/I [26] cause particular problems because of the antiquated aspects and features of these languages. Here we focus on Visual Basic (VB) and COBOL, because of the widespread use of these languages for business-critical finance and banking systems [33]. Translation of VB to Python is relevant as the finance industry is increasingly using machine learning, natural language processing (NLP) and other artificial intelligence (AI) techniques, for which there is strong Python support. Translation of COBOL to Java or C# facilitates the integration of legacy systems into modern software environments and platforms. We also apply reverse and re-engineering to analyse and remove quality flaws from legacy Python systems.

2.1 BASIC and variants

BASIC[4] was intended, as its name suggests, as a language for inexperienced programmers to use for relatively simple programming problems. It became popular with the advent of home PCs in the 1970s, and as Visual Basic (VB) and Visual Basic for Applications (VBA) became the main language for defining auxiliary code modules within Microsoft (MS) applications such as Excel [50]. The last version of VB was VB6 in 1998.

Key challenges for software modernisation and re-engineering of VB/VBA are:

- The use of implicit typing for data items
- GOTO statements
- The large number of different kinds of statements (there are 67 statement kinds in VB6)
- The complexity of MS applications such as Excel, with complex spreadsheet data and hundreds of application functions, which can be called from VBA code.

As a case study in VB6/VBA analysis we consider a legacy bond pricing system, which originated in the 1990s and is still in use. The dependence of VBA code on MS application data and functionality means that the code cannot be understood in isolation from the host application. For example, the bond case study includes the code:

```
B1 = Sheets("FBU").Range("m20")
B2 = Sheets("FBU").Range("m27") -
        Sheets("FBU").Range("m20")
```

whose meaning depends upon the semantic denotation of cells M20 and M27 in the referenced worksheet. In this case

[4] Beginner's All Purpose Symbolic Instruction Code.

there was also insufficient documentation in the worksheet regarding the semantics.

2.2 COBOL

COBOL (c. 1959) predates BASIC, and was specifically intended as a language for commercial data-processing applications. The main standard versions are COBOL '74 and '85. Legacy COBOL presents a wide range of challenges for software modernisation/re-engineering:

- No general data types—instead data items are assigned a specific format at the byte level
- Extensive use of aliasing and redefinition of data items
- GO TO and ALTER statements (an ALTER statement changes the destination of a GO TO at runtime)
- A primitive procedure call mechanism, PERFORM, which is vulnerable to many control flow anomalies [14]
- Multiple variants and formats of statements.

Because of the long history of use of COBOL, legacy COBOL systems may bear the imprint of archaic coding practices based on assembly programming and primitive editing facilities, with GOTO-based control flow and many embedded literals [67]. The industrial billing system which we considered in [38] had already been modernised to COBOL '85 and the use of PERFORM calls prior to our re-engineering work, but it still retained essentially the same GOTO-based control flow of an earlier COBOL '74 version.

A COBOL program is structured into four main parts: (i) an *identification division*, (ii) an *environment division* (defining in particular the link between data storage devices and logical files), (iii) a *data division* (defining all program data) and (iv) a *procedure division* (defining program behaviour). In turn, each division is divided into sections, and in the procedure division, further divided into *paragraph*s, *sentence*s and *statement*s. Paragraphs are a key program unit, consisting of a name label and a sequence of sentences. Paragraphs may be both the target of a GO TO, and invoked via a PERFORM. Thus at least two separate semantics need to be expressed for each paragraph P:

1. The semantics encountered by invoking P by $PERFORM\ P$
2. The semantics encountered by a $GO\ TO\ P$, or by 'falling through' into P from a preceding paragraph.

2.3 Python 2 and Python 3

Python has become one of the most popular programming languages both for general use and for machine learning and other AI applications. Its relative simplicity compared to Java and C# has enabled its use by a wide range of people, without needing specialised programming skills, indeed it plays a similar role to the original BASIC language in this respect. However this simplicity comes at a cost, in particular the lack of explicit variable typing and explicit variable declarations can lead to unclear and difficult-to-maintain programs.

The main problem areas with the maintenance of Python applications are:

- *Implicit typing*: the type of a variable is not fixed, and depends on the data that is held in the variable. Data of different types can be held in one variable at different times, leading to error-prone and confusing code.
- *Implicit declarations*: local variables are implicitly declared by the first assignment to them (at runtime). Moreover declarations persist from an inner scope to an outer scope, contrary to block-structured languages such as Java. For example, the declarations of y in the following code are valid in Python:

```
def f(x) :
 if x < 0 :
   y = -x
 else :
   y = x
 return y
```

- *No interfaces*: Only classes can be defined.
- *Procedural programming*: the language can be used in the same procedural manner as VB, with programs coded as a monolithic list of functions. Thus similar flaws such as excessive parameter lists (EPL), Table 4, can arise due to a lack of encapsulation of data in objects. Ad-hoc lists and tuples of heterogeneous data items may be used instead of objects to pass and return groups of data from functions.
- *Multiple/non-standard inheritance*: The Python version of inheritance differs from standard OO inheritance as found in languages such as Java and C#, in particular a form of multiple inheritance with prioritised overriding is supported.
- *Language evolution*: Python changed substantially from Version 2 to Version 3, and these versions have incompatible syntax and semantics. For example, the $print$, $raise$ and $except$ statements have different syntax in the two versions, and integer division is denoted differently. It is a non-trivial task to migrate large systems from Python 2 to Python 3 [62].

3 Program abstraction using ANTLR and $\mathcal{CSTL}$

ANTLR is a parser generator which takes as input a grammar file $LParser.g4$ for a software language L, and produces an executable parser for L. It has been widely-used for the definition and processing of diverse software languages, and has an active support community. Here we utilise the VisualBasic6.g4, Cobol85.g4 and Python.g4 gmars from antlr.org. ANTLR grammar rules have forms such as

```
tag:
 tag1 sym tag2
```

whereby a grammar non-terminal category tag is defined based on other non-terminals $tag1$, $tag2$ and terminal symbols sym, etc. ANTLR parsers produce parse trees with the textual form (tag t_1 ... t_n) where tag is the name of the grammar rule which produced the tree, and the t_i are the immediate subterms/subtrees.

$\mathcal{CSTL}$ is a text-to-text (T2T) transformation language, based on pattern-matching and text substitution. It has been used for specifying code generators and other language-to-language mappings [41, 43]. In this paper we use $\mathcal{CSTL}$ to abstract COBOL, VB6 and Python programs to UML/OCL.

$\mathcal{CSTL}$ uses concrete syntax templates for matching source elements, in addition to concrete syntax templates for constructing target text. The left hand side (LHS) of a $\mathcal{CSTL}$ rule

```
LHS |--> RHS
```

is written in a schematic concrete syntax for the source language, and the right hand side (RHS) is expressed in schematic concrete syntax of the target language. Metavariables $\_i$ for $i : 1..99$ can be used on both the LHS and RHS, on the LHS they denote some source language item (a term in a parse tree according to the source language grammar) and on the RHS they denote the concrete textual syntax of the translation of that item. For example, the rule:

```
if _1 then _2 else _3 endif
 |-->(_1)?_2:_3
```

defines translation of OCL conditional expressions to Java. Rules may have conditions <when> $Cond$, and actions <action> Act, written after the RHS. Rules are grouped into rulesets, corresponding to grammar rules for the source language.

For COBOL ‘85 the $\mathcal{CSTL}$ rules for abstraction of ADD statements include:

```
addStatement::
ADD _1 END-ADD  |-->_1<when> _1 addToStatement
ADD _1  |-->_1<when> _1 addToStatement

addToStatement::
_1 TO _2  |-->
   _2 := (_2 + _1)_2`roundFunction ;\n
_1 _* TO _3  |-->
   _3 := (_3 + _1 + _*`sum)_3`roundFunction ;\n

addFrom::
_1  |-->_1

addTo::
_1 ROUNDED  |-->_1
_1  |-->_1

roundFunction::
_1 ROUNDED  |-->->roundTo(_1`fractionWidth)
_1  |-->->truncateTo(_1`fractionWidth)
```

$addStatement$, $addFrom$, $addTo$ and $addToStatement$ are grammar rulenames (non-terminals) of the Cobol85.g4 gmar. sum is a built-in $\mathcal{CSTL}$ function to produce a $+$ sum of a list of terms, and $roundFunction$ is a custom function defined by the final ruleset above. The rule condition _1 $addToStatement$ restricts the rule to apply only in the case that the _1 metavariable is bound to a parse tree with tag $addToStatement$.

Metavariables $\_i$ can match against a single source term, and $\_*$ can match a list of one or more terms. The notation `_i`f` denotes the application of ruleset or function $f$ to the item bound to variable $\_i$, likewise for `_*`f`. $fractionWidth$ is the number of fractional digits in the assigned variable, which is determined when processing its declaration. $roundFunction$ is also used for other COBOL arithmetic statements.

A COBOL statement `ADD P Q R TO VX ROUNDED` will match against the LHS templates of the $\mathcal{CSTL}$ rules for $addStatement$ and $addToStatement$ (the second rules in each ruleset). P will be assigned to _1 in the $addToStatement$ rule, the list $[Q, R]$ to $\_*$, and the parse tree term ($addTo$ VX $ROUNDED$) to _3.

The $addToStatement$ rule then produces the output text

```
VX := (VX + P + Q + R)->roundTo(N) ;
```

expressing the semantics of the COBOL statement, where N is the number of fraction digits in the data format of VX, by applying the rules for $addFrom$, $addTo$, sum and $roundFunction$ and substituting the results into the RHS template of the rule.

A key advantage of using $\mathcal{CSTL}$ is that the abstraction and code-generation scripts can be directly modified and extended by end-users to customise scripts to address specific aspects of their legacy code recovery task. Specialised MDE knowledge is not needed, only an understanding of the source language grammar and of text matching/substitution. Scripts can be executed from the command line by using the ‘cgtl’ command. No compilation of scripts is necessary.

Representation of program semantics in AgileUML uses extensions of the OCL 2.4 standard [55], in particular a procedural extension of OCL with a Pascal-like syntax, similar to the SOIL formalism [9], is used to represent procedural statements. The extended OCL statements can be used to define the effect of an operation by an operation *activity* in addition to or instead of an operation postcondition.

Libraries *OclFile*, *OclDate*, *OclProcess*, *OclRandom*, *OclType* and *MathLib* for common programming data types and facilities have also been defined for AgileUML [42]. These libraries can be directly edited or extended by users of our approach.

For COBOL, VB6 and Python we added further library operations, types and components:

- The *FinanceLib* library component containing operations for financial functions such as (net) present value, annuity, internal rate of return, etc.
- Operators $\rightarrow roundTo$ and $\rightarrow truncateTo$ generalising $\rightarrow round$ and $\rightarrow oclAsType(int)$.
- An *OrderedMap* data structure, which enables access to values both by key and by a positional index. This can also be simulated by the type $Sequence(Map(String, OclAny))$.
- For VB, a library class *Excel* to represent the data and functions of Excel spreadsheets.
- For Python, specific string formatting operations and matrix operations, in libraries *StringLib* and *MatrixLib*.

3.1 VB6/VBA

Table 1 summarises the program abstraction strategy for VB6/VBA code.

In general the structure of the source code is retained in the abstraction, facilitating traceability.

In the cases of typeless variable declarations

```
DIM Var
```

type inference based on the values assigned to *Var* and the processing performed on it are used where possible to deduce the specific type of *Var*. In the absence of specific typing knowledge, the ‘universal’ OCL type *OclAny* is used for *Var*.

3.2 COBOL ‘85

Table 2 summarises the abstraction strategy for COBOL ‘85.

The data parts of a COBOL program *Prog* are abstracted as follows:

- Data declarations in the file section, linkage section and working storage section are represented as attributes of a class *Prog_Class* representing the COBOL program.
- The OCL type of a data item is obtained from its COBOL format (the PIC/PICTURE format specifier). For example, format 9(4) is represented as *int*, 9(14) as *long*, and 99V99 as *double*. Composite items do not have an explicit PIC/PICTURE, and are given a *String* type.
- Implicit or explicit aliasing relations between data items are expressed as *invariants* of *Prog_Class*. For example, the declaration

```
01 REC.
 02 FLD1 PIC XX.
 02 FLD2 PIC 9999.
```

is expressed as:

```
attribute REC : String;
attribute FLD1 : String;
attribute FLD2 : int;

invariant REC = "" +
 StringLib.leftAlignInto(FLD1, 2) +
 StringLib.padLeftWithInto(FLD2, "0", 4);
invariant FLD1 = REC.subrange(1,2);
invariant FLD2 =
 REC.subrange(3,6)->toInteger(10);
invariant 0 <= FLD2 & FLD2 <= 9999;
```

Thus the field data `FLD1 = "T"` and `FLD2 = 100` corresponds to `REC = "T 0100"` with a space between *T* and 0100.

As noted in Sect. 1, we use the AgileUML toolset to produce code in target programming languages. A feature of AgileUML is that its inbuilt code generation strategy (for Java, C# and C++) enforces invariants $P \Rightarrow Q$, by synthesising update code for Q where possible so that any event which makes P true also establishes Q [39]. Thus the use of invariants enables the source code semantics of data relations to be correctly expressed in the abstraction, in a manner that can be implemented in forward engineering.
- Files are abstracted as sequences together with a file position pointer. Indexed files also have an index map from file key values to file positions.

The procedure division of program *Prog* is abstracted as follows:

- Each paragraph P is represented by two separate operations $P\_Call()$, $P()$ of *Prog_Class*. $P\_Call()$ represents the functionality of a $PERFORM\ P$ statement, whilst $P()$ represents the fall-through/$GO\ TO\ P$ functionality. $P()$ invokes $P\_Call()$:

```
operation P()
pre: true post: true
```

Table 1 Abstraction mapping from VB6/VBA to UML/OCL

VB6/VBA construct	Abstraction
module	package and main class
record	class with ≪*struct*≫ stereotype
record field	attribute
function, subroutine	operation
enum definitions	enumerated types
assignment, sequencing	assignment, sequencing
blocks	blocks
calls of functions subroutines/if, while, return	operation calls if, while, return
do, for-next	while
for each in select	for
error	if
exit loop	error break
Integer, Byte, Long	int
LongLong	long
Single, Double String	double
Array types	String
Collection	Sequence types Sequence(Map(String,OclAny))

Table 2 Abstraction mapping from COBOL ‘85 to UML/OCL

COBOL ‘85 construct	Abstraction
Program suite	package
Program	class
record	class + invariants
record field/program variable	attribute
section, paragraph, subprogram	operation(s)
arithmetic statements, move, set	assignment
sequencing, in-line performs	sequencing, blocks
calls of subprograms, simple	operation calls
PERFORM, PERFORM THRU GO TO IF, EVALUATE	if
PERFORM UNTIL/varying, exit	while, return
simple PERFORM TIMES	for
GO TO DEPENDING ON	if + calls
on size error	tests for overflow conditions
inspect, string, unstring	String/regex operations
Merge/Sort	sorting operations
stop run	OclProcess.exit(0)
Numeric/computational data formats	int/long/double
Alphabetic/alphanumeric formats	String
Array (table) types	Sequence types
Files	OclFile, sequences, maps

```
activity:
 self.P_Call() ;
 self.Q() ;
```

where Q is the next paragraph following paragraph P, if any.

- A section is also represented by two operations in the same manner.

Additional paragraph representations may be necessary if the $PERFORM\ A\ THRU\ B$ variant of PERFORM is used: each paragraph P from A down to B would also have a version P_thru_B:

```
operation P_thru_B()
pre: true post: true
activity:
```

```
self.P_Call();
self.Q_thru_B();
```

where B_thru_B is simply B_Call.

3.3 Python 2/Python 3

Table 3 summarises the abstraction strategy for Python versions 2 and 3.

Python programs are a list of items, which may either be statements, function definitions or class definitions. Function definitions contain a list of statements, and class definitions contain a list of statements and functions. This organisation can be mapped to a UML specification which has a class for each Python class, and in addition a class $FromPython$ containing the top-level functions as its operations and the top-level statements as its initialisation code.

Explicit declarations of variables are inferred from the implicit Python declarations, and type inference is used to identify variable and parameter types where possible. For example, if a parameter p is used in a numeric computation $math.pow(p, x)$, then its type can be inferred as $double$.

4 Quality analysis and quality improvement

Once a source application has been abstracted to UML/OCL, it can be analysed for quality flaws (Sect. 4.1) and restructured to improve its quality. Performing restructuring at the specification level avoids the need to define multiple language-specific restructuring tools, and means that the improved structure applies to every target platform. Both localised restructuring of individual operations and classes (refactoring, Sect. 4.2) and global restructuring of a system at the architectural design level (rearchitecting, Sect. 4.3) can be carried out.

Other kinds of restructuring could also be applied, such as transforming procedural code patterns such as *Sieve* into object-oriented design patterns such as *Chain of responsibility* [38]. However such transformations involve extensive structural changes, which can impact on traceability and on testing effort, and we do not currently support these.

4.1 Quality analysis

AgileUML provides analysis of UML/OCL specifications and designs to detect flaws analogous to ‘code smells’ in the sense of [19, 25]. These can also be regarded as technical debt indicators [10, 48, 69]. Table 4 shows the flaws we detect and the corresponding refactorings that can be used to address these flaws.

The thresholds for these flaws (e.g., a minimum size for clones) can be varied depending on the practices and internal standards of the client organisation. The *flaw density* of a software artefact is the count of flaws, divided by the size of the artefact, typically given in lines of code (LOC).

Commented-out code lines and self-declared technical debt can also be indicators of maintenance problems [17, 70].

4.2 Refactoring

Some detected flaws can be removed or reduced by refactoring the specification, for example by splitting a large operation into parts to reduce EFS and EFO. EPL can be reduced by factoring out a group of 2 or more related parameters as a ‘value object’ class VO and replacing the group by a single $vox : VO$ parameter. If several operations have the same VO parameter class after this step then these operations can be moved into the VO class.

The abstraction process described in Sect. 3 replaces GO TO jumps by function/operation calls, resulting in structured code which however may use recursion (introducing a CBR_2 flaw). Tail recursion can be removed by the ‘replace recursion by iteration’ code transformation, which applies both to operations defined by OCL expressions and those defined by statements.

Expressed in $\mathcal{CSTL}$, the restructuring rules for this transformation include:

```
operation _1 ( )\n
activity: _* ; self._1 ( ) ; |-->
 operation _1()\n
 activity: while true do ( _* );\n\n
```

Mutual recursion can be reduced to self-recursion by using the ‘Replace call by definition’ refactoring.

CBR_1 without CBR_2 indicates that there are more non-cyclic dependencies between operations than the number of operations. In this situation it is possible to partition the set of operations into groups where operations in a ‘higher level’ group call operations of a ‘lower level’ group but not vice-versa.[5] The partitions then form the basis for splitting the original class into new client/supplier classes. Likewise if there are excessive numbers of operations in a class (ENO) the class should be split where possible based on operation calling relations or on a conceptual basis.

DC can be reduced by factoring out the cloned code into a new operation, replacing the duplicate occurrences by a call to this operation. DE can be addressed by introducing a

[5] Select any operation *op* with outgoing calls. The set G_1 of operations called directly or indirectly from *op* is disjoint from the set G_2 consisting of *op* and its direct/indirect callers. No operation in G_1 can call an operation of G_2.

Table 3 Abstraction mapping from Python 2/3 to UML/OCL

Python construct	Abstraction
program	package and main class
class	class
single inheritance	inheritance
multiple inheritance	inheritance + overriding
global variable	attribute of main class
local variable	local variable
async function	run operation of $\ll active \gg$ class
function	operation
assignment	assignment/declaration
sequencing, blocks	blocks
function calls	operation calls
object creation $x = C()$	Call $x := C.newC()$
if	if
for	for
yield	defines $next$ for OclIterator representing the generator
raise, try, except	error, try, catch
assert	assert
with	try/catch
int/float	int/double
str/bool	String/boolean
list, tuple types	Sequence types
matrix types	Nested sequences
dict type	Map type
set type	Set type
files	OclFile

new local variable, initialised to the cloned expression, and replacing other occurrences of the expression by the new variable. UOP and UVA can be reduced by removing the unused elements. MGN is addressed by defining new constants with the literal value and replacing occurrences of the literal value by the constant name.

PMV and MDV typically arise in Python coding because of the absence of explicit local variable declarations in Python. Instead variables are implicitly declared by the first assignment to them encountered during execution. This can lead to code that is unclear and difficult to debug or maintain. We relocate and merge declarations using the ‘hoist local declarations’ refactoring to remove multiple declarations of a variable where possible. For example, the code:

```
def f(x) :
 if x < 0 :
   y = -x
 else :
   y = x
 return y
```

is abstracted and refactored to:

```
operation f(x : double) : double
pre: true post: true
activity:
 var y : double := 0.0;
 if x < 0 then y := -x
 else y := x;
 return y;
```

The refactorings supported by the current version (2.3) of AgileUML are given in Table 5.

Some of these refactorings (Split class, Move operation to parameter class, Extract value object) are particularly relevant for transforming a procedural program structure into an object-oriented specification. Classes are also introduced by the rearchitecting process described in the following section.

4.3 Rearchitecting

Rearchitecting attempts to produce an improved software architecture for a legacy system. This can involve segre-

Table 4 Design flaws and refactorings

Flaw	Name	Description	Refactorings
ENO	Excessive number of operations	More than 10 operations in a class	Split class
EFS	Excessive operation/function size	Length $>$ 100 LOC	Split operation
EPL	Excessive parameter length	More than 5 parameters of a function/operation	Extract value object; move operation to parameter class
CC	High cyclomatic complexity	Cyclomatic complexity ≥ 5	Split operation
EDN	Excessive nesting depth	Nesting depth of statements ≥ 5	Reduce code nesting
EFO	Excessive fan-out	$\geq$ 5 different functions/operations called from one function/operation	Split operation
EFI	Excessive fan-in	$\geq$ 5 different functions/operations call one function/operation	Split class
CBR_1	Coupling bound 1	Size of call graph $>$ number of operations	Split class
CBR_2	Coupling bound 2	Cycles in call graph	Replace call by definition; replace recursion by iteration
DC	Duplicated code	Duplicated segments of code	Extract operation
DE	Duplicated expression	Duplicated complex expressions within one operation	Extract local variable
UOP	Unused operations/functions	Operations that have no callers	Remove operations
UVA	Unused variables	Variables/parameters that are never read/written	Remove variables
MGN	Magic numbers	Literal constant values in statements	Replace literal by named constant
PMV	Polymorphic variable	Same variable assigned values of different (incompatible) types	Define separate variables
MDV	Multiply-declared variable	Declarations of 2+ same-named variables within same scope	Hoist local declarations

Table 5 Design refactorings

Name	Effects
Extract interface	Defines new interface for class, with declarations of its operations
Extract operation	Defines new operation for cloned statements
Extract local variable	Defines new local variable for duplicated expression evaluations
Replace literal by named constant	Replace duplicate literal value expressions by new named constant
Split class	Splits class into clients/suppliers
Remove multiple inheritance	Combines multiple superclasses into single superclass
Move operation to parameter class	Moves operation to class of first class-typed parameter
Replace call by definition	Inlines operation definition at point of call
Replace recursion by iteration	Changes tail-recursive operation to iterative version
Split operation	Creates new operation for subsection of operation activity
Hoist local declarations	Moves all declarations of one variable to start of activity
Reduce code nesting	Replace nested conditional statements by a sequence of conditionals
Extract value object	Create class for group of parameters used by multiple operations

gating platform-specific code from business logic code, and replacing direct calling dependencies between components by indirect connections via interfaces.

Typically, legacy VB6 and COBOL programs are defined in a monolithic architectural style, with a single module/program unit containing all code elements as a collection of operations/functions, together with global scope data definitions. Legacy Python programs may also have a procedural structure as a list of global functions and statements, without class definitions. In order to identify meaningful subcomponents of these programs, we analyse the calling dependencies of operations on each other (the program call graph) and the dependencies of operations on global data items.

This information can be used to automatically modularise a system according to metrics of cohesion and coupling between components, using an optimisation procedure [8]. There are two main measures: of *coupling* and *cohesion*. $Coupling_{i,j}$ measures how many data and calling dependencies exist between different components C_i, C_j, whilst $Cohesion_i$ measures how many dependencies exist within an individual component C_i. Here we only consider direct calling dependencies between operations:

$$(1)\quad Cohesion_i = \frac{Calling\ dependencies\ in\ C_i}{N_i * (N_i - 1)}$$

where $N_i > 1$ is the number of operations in C_i.

$$(2)\quad Coupling_{i,j} = \frac{Calling\ dependencies\ between\ C_i,\ C_j}{N_i * (N_j - 1)}$$

for $N_i > 0$, $N_j > 1$.

The optimisation process tries to maximise the *class responsibility assignment index* (CRA-Index) measure defined as:

$$CRA{-}Index = \Sigma_i\ Cohesion_i - \Sigma_{i,j}^{i \neq j}\ Coupling_{i,j}$$

We implement CRA optimisation and component identification via a set of heuristic rules:

1. An operation f is placed in the same component as operation g if g is the only caller of f and g is itself called by some operation.
2. Operations with no caller remain in the parent/original component.
3. Operations in the same cycle of call dependencies are placed in the same component.
4. Other unallocated operations are placed in a component which has a maximal number of calling dependences on the operation.
5. Components with a common element are merged, as are mutually-dependent components.
6. Data items which are only referenced by one component C are placed in C, otherwise the item is placed in a data layer component, which is depended-on by all components that use global data. The data layer has the responsibility to maintain data integrity constraints such as the invariants derived by COBOL abstraction (Sect. 3.2).

Figure 2 shows an example of applying the heuristics to create components $C0$ to $C3$ from an unstructured system with 9 operations. An arrow $e \longrightarrow f$ denotes that operation e has a direct calling dependency on f. Identified components are marked by red rectangles. The components will be defined as UML classes containing the specifications of the component operations. The CBR_1 flaw of the original ver-

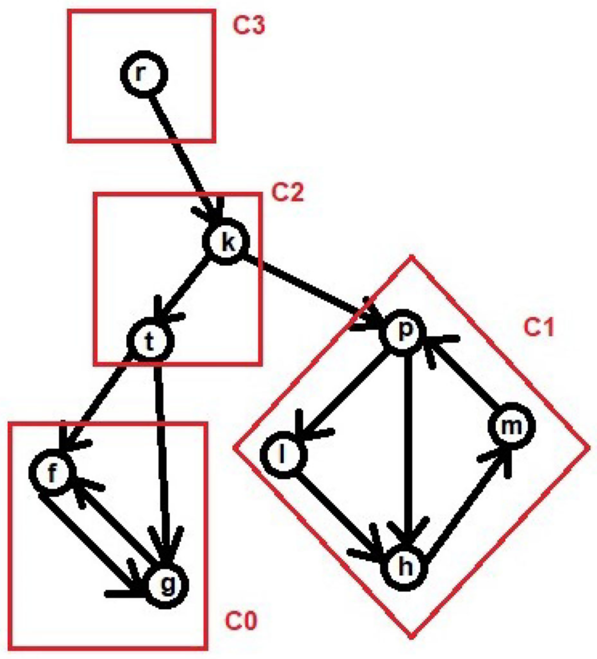

Fig. 2 Rearchitecting example

sion (12 direct calling dependencies between 9 operations) is then removed. Only $C1$ has this flaw in the restructured system. CBR_2 is also reduced from 6 in the original component to a maximum of 4 in $C1$.

Each identified component C can be further transformed by extracting an interface CI which declares the externally-used operations of C. Client components then use C's operations via CI. For example, component $C1$ in Fig. 2 has the provided interface $C1I$ which declares the single operation p.

A significant hindrance to maintenance is the intermixing of business logic code together with data management code using a specific technology, e.g., direct invocation of Excel or SQL commands within business logic [49]. This design flaw leads to high maintenance effort/costs when there is technology change, such as a move from a relational to a NoSQL database. As a result of the abstraction process, file and database operations are represented by calls of the library components $OclFile$ and $OclDatasource$, and internet/socket communications are also represented as $OclDatasource$ calls. Excel operation calls are expressed by calls of the $Excel$ library operations. It is therefore direct to identify all such processing and to segregate the code by defining new *data access object* (DAO) components, which encapsulate the specific processing. The DAO operations are then called from the business logic components. A *required interface* C_DAO_Req of each business logic component C is extracted, which expresses precisely which DAO operations are required by C.

The resulting architecture will normally satisfy the key principles of the clean architecture [49]:

- *No cyclic dependencies between components (i.e., the ADP)*: this holds due to heuristic rules (3) and (5)
- *Interface segregation principle (ISP) "Don't depend on things you don't need"*: this holds because each component Ci only depends on another component Cj if some operation of Ci calls an operation of Cj. This dependency is restricted to a required interface Ci_Cj_Req of Ci, which may be a strict subset of the provided interface CjI of Cj.
- *Dependency rule (DR) "Platform-independent components should not depend on platform-specific components"*: as described above, required interfaces DAO_Req of business tier components are introduced, which the DAO components for technology-specific resource management then depend upon (their provided interfaces must satisfy/extend the DAO_Req interfaces).

The *Single responsibility principle (SRP)* of [49] – that each component is only used by one actor—is only ensured if this holds for the original program.

5 Semantic preservation

A key requirement of businesses using re-engineering is that the source program functionality should be retained accurately in the target [16, 67]. Moreover, it should be direct to confirm this by relating parts of the source and target. We address these concerns by:

1. Showing that the abstraction process preserves system semantic properties (Sect. 5.1).
2. Showing that refactoring steps also preserve semantics (Sect. 5.2).

It is then the responsibility of the utilised forward engineering technology to preserve semantics and code structure from UML/OCL to the target. AgileUML has an established track record of use for Java and C# code generation over 20 years, whilst for Python it has been used extensively for over 10 years. Thus there are good assurances of correctness for generation of these target languages.

5.1 Semantic preservation by abstraction

The semantics of OCL 2.4 is defined by Annex A of [55] in terms of the mathematics of Zermelo-Fraenkel set theory (ZFC) [12]. OCL types T are given a mathematical denotation $I(T)$ as a set of elements, for example

$$I(Integer) \;=\; \mathbb{Z} \cup \{\varepsilon, ⋔\}$$

where ε is the denotation of the *null* OCL element, and ⋔ the denotation of the *invalid* OCL element. AgileUML uses computational subsets *int*, *long* and *double* of the mathematical datatypes *Integer* and *Real* of OCL. These can be given denotations as follows:

$$I(int) = \{x : \mathbb{Z} \bullet x \geq -2^{31} \; and \; x \leq 2^{31} - 1\}$$
$$I(long) = \{x : \mathbb{Z} \bullet x \geq -2^{63} \; and \; x \leq 2^{63} - 1\}$$
$$I(boolean) = \{false, true\}$$

$MAXINT$ is the upper bound $2^{31} - 1$ of the *int* range. The denotation of *double* is the IEEE 754 64-bit floating point range. The denotation of *String* is all finite sequences of characters from an alphabet *A* assumed to include the ASCII characters numbered 0 to 127.

Similarly, expressions *expr* of OCL can be given a denotation $I(expr)$ in ZFC. Literal integer numeric values are written without a decimal point, whereas literal doubles have the decimal. However, corresponding integers and doubles are considered equal, e.g.: $3 = 3.0$. The relational operators are otherwise given their standard denotations, with lexicographic ordering used as the standard order relation $<$ for strings.

The definition of arithmetic operators $+, -, *, /, div$ and *mod* follows the usual mathematical interpretation of these operators (subject to the IEEE 754 approximations for double computations), with the proviso that the result is considered unspecified if the usual mathematical result would fall outside the numeric range concerned. Thus $1 + MAXINT$ is meaningless as an integer computation. Likewise, division by zero is not specified.

As an example of semantic definition, integer addition on *int* is defined as:

$$\begin{aligned} I(e_1 + e_2) = I(e_1) + I(e_2) \\ when \; I(e_1) \in I(int), \quad I(e_2) \in I(int), \\ I(e_1) + I(e_2) \in I(int) \end{aligned}$$

We can then argue that integer numeric computations in the source program, which do not result in overflow/underflow outside of their defined ranges, and do not involve division by zero, are translated correctly to corresponding OCL expressions. This assertion relies upon the source programming language/environment correctly implementing such integer computations.

Note that restricted program subtypes of *int*, such as Java *short* or VB *Integer* (16-bit integers), are abstracted to *int* in OCL. In such a case there could be overflow within the program type that would not occur in the abstracted type, however such a case is not included in our correctness assurance. Especially in the case of COBOL, all integer numeric data has a specific length in terms of decimal digits, and we only assure that computations that remain within the implied restricted types are abstracted correctly to OCL. The COBOL semantics in many cases of numeric overflow are unspecified [13].

String and boolean operators such as *toUpperCase* and *xor* have precise definitions in OCL and it is generally possible to map program string and boolean operators to equivalent representations in OCL which use the OCL operators.

For real-number computations, provided the source language/platform conforms to IEEE 754 floating point for 64-bit values, the arithmetic operators +, −, *, / will have the same semantics on *double* values at source and abstraction levels, provided that exceptions do not occur (e.g., evaluating $log(0.0)$) and provided that no computation produces values outside of the *double* range. The VB *Double* type and Python *float* type conform to the IEEE 64-bit standard. In contrast, COBOL uses fixed-point representations for real numbers, where each numeric format has a fixed number of fractional digits after the decimal point. In order to correctly express COBOL real-number numeric semantics, we introduced new OCL operators $\rightarrow roundTo$ and $\rightarrow truncateTo$, which round/truncate *double* values to a given number of decimal places.

Examples of abstraction mappings for COBOL numeric and string expressions include:

```
functionCall::
FUNCTION ACOS ( _1 ) |-->(_1)->acos()
FUNCTION EXP ( _1 ) |-->(_1)->exp()
FUNCTION FACTORIAL ( _1 ) |-->
      MathLib.factorial(_1)
FUNCTION LOWER_CASE ( _1 ) |-->
      (_1)->toLowerCase()
FUNCTION MAX ( _1 _* ) |-->
      Set{_1_*'arguments}->max()
```

As described in Sect. 3.2, COBOL data definitions also imply the existence of invariant relations between different variables, and these invariants must be maintained by the abstracted version of the source code, and by any target implementation.

Collection types and arrays are *reference* types in VB, COBOL and Python: a value of such a type used as a parameter to a procedure will be passed by reference. In contrast, in standard OCL the collection types are value types and are passed by value. To improve efficiency, in AgileUML they are however treated as reference types, so that the AgileUML collection semantics coincides with program semantics for

VB, COBOL and Python. Class types are reference types in standard OCL and AgileUML, as with class types in Python and records in COBOL. Record types are value types in VB, so they are abstracted to classes marked with the stereotype $\ll struct \gg$ to indicate that value semantics should be used (as with structs/record types in C and Pascal). This means that instances of such classes will be copied when assigned and passed as parameter values.

The semantic preservation of statement semantics by the abstraction mapping follows directly from the preservation of expression semantics, due to the close correspondence of program statement structures and extended OCL statement structures, at least for the statement forms listed in Tables 1, 2, 3. It is assumed that there is no aliasing between variables other than that due to data definitions (in the case of COBOL) and parameter passing by reference. It is also assumed that ALTER statements are not used in COBOL code, and that PERFORM anomalies such as overlapping PERFORM-THRU ranges do not occur, i.e., it is assumed that PERFORM calls can be treated as normal procedure calls.

The inbuilt functions and operations of COBOL, VB6 and Python are handled by the abstraction process. There is also coverage of over 100 Excel worksheet functions in the case of VB6. For Python, there is at least partial coverage of the external libraries *re*, *pickle*, *datetime*, *asyncio*, *numpy*, *random* and *math*, but not of other external libraries.

5.2 Semantic preservation by refactoring steps

We also need to show that any refactoring and rearchitecting actions also preserve semantics, in general this will be the case because these actions only reorganise data and functionality, but do not change the observed semantics. For example, class splitting moves operations from the original class into new classes.

Table 6 defines the conditions under which refactorings preserve semantics.

Considering in particular the 'Extract local variable' refactoring, the original activity will have a structure such as:

```
statements1 ;
statements2[expr] ;
statements3 ;
statements4[expr]
```

where there are at least 2 occurrences of the same complex expression $expr$, and the variables/attributes used in $expr$ are not written by any of the statements in the activity.

The refactored version will have the structure:

```
var v : T := expr ;
statements1 ;
statements2[v] ;
statements3 ;
statements4[v]
```

where v is a new variable name, T is the type of $expr$, and the $expr$ occurrences are systematically replaced by v throughout the activity. It is therefore clear that the correctness condition for this refactoring does ensure the preservation of the activity semantics.

The 'Replace recursion by iteration' refactoring is more complex. This is restricted to operations of the form:

```
operation op(pars) : T
activity:
 statements;
```

where each control flow branch in $statements$ ends in either:

1. A recursive call `return op(vals)`
2. A statement `return val` where *val* involves no direct or indirect call of *op*
3. A program exit `OclProcess.exit(n)`

The refactored version has the form:

```
operation op(pars) : T
activity:
 while true do
   statements1;
```

In this version, the recursive calls `return op(vals)` of `statements` are replaced in `statements1` by simultaneous assignments `pars := vals` followed by `continue`, and the other forms of control flow branch end are left unchanged.

We can argue that the refactored version has the same functionality as the original, by induction on the depth of recursive calls which arise in an execution of the original *op* version. In the base case where no recursive call occurs, a branch of form 2 or 3 above is executed, in both `statements` and `statements1`, and this terminates the `while` loop and execution of *op* in the restructured version, with the same effect as in the original version.

If the execution of *op* leads to a recursive call via a branch of type 1 in `statements`, then in `statements1`, the actual parameters *vals* are assigned to the formal parameters $pars$, and the next iteration of the `while` loop starts. This is the same as calling the restructured *op* as $op(vals)$, and hence

Table 6 Refactoring correctness conditions

Name	Condition
Extract interface	Always applicable
Extract operation	New operation *ex* is in same class as original operation *op*. Cloned code can write attributes, not local variables of *op*
	Parameters of *ex* are the variables/parameters of *op* read by the cloned code
Extract local variable	Expression value is same at each location in the extraction scope
Replace literal by named constant	Name must be new
Split class	$\mathrm{CBR}_1 > 0$ with $\mathrm{CBR}_2 = 0$
Remove multiple inheritance	No clashing features in the superclasses
Move operation to parameter class	Operation only reads parameters/entity parameter attributes; only writes entity parameter attributes
Replace call by definition	Features accessed directly in called operation may need to be accessed via getters/setters in definition expanded inline
Replace recursion by iteration	Operation is tail recursive; not included in any mutual recursion
Split operation	As for Extract operation
Hoist local declarations	The multiple declarations must be of inter-convertible types
Reduce code nesting	Program logic must be preserved
Extract value object	Parameter group should have consistent semantics

by induction, this results in exactly the same functionality as a call $op(vals)$ of the original op version.

Thus, for any terminating call of the original version of op, the restructured version also terminates and produces the same result/effect.

Similar reasoning can be applied to confirm that the other refactorings of Table 5 also preserve system semantics under the conditions of Table 6.

With regard to rearchitecting, the heuristics and restructurings described in Sect. 4.3 can be considered as architectural transformations, such as *Extract Interface* and *Extract DAO*. Each of these can be shown to preserve system semantics, because they do not change the details of operation definitions, but only the boundaries of components.

6 Specification extraction

In this section we describe techniques for deriving or extracting high-level requirements specifications from code and from abstracted system models.

In mainstream re-engineering practice, the process of relating legacy code data and functionality to business requirements typically involves consultation with system experts to build up a consistent and systematic understanding of the system in business-specific terms [16, 67]. We have also used this approach (for example, with the VB industrial case) when domain experts are available. However this approach has limitations, because the system experts may only have expertise gained from maintaining the system, and the knowledge of the original developers and stakeholders of the system is no longer available. Thus in-depth understanding of the reasons behind design decisions has been lost, and such reasons can only be inferred.

Another approach to derive specification-level documentation is to use automated code summarisation, whereby natural language explanations are generated from program code. Code summarisation techniques have mainly focussed upon modern programming languages, and there is a lack of datasets and tools for COBOL and VB [71]. The advent of large language models (LLMs) such as the GPT series [72] with extensive natural language and programming knowledge represents another means to obtain code summarisations. We tested GPT 3.5 (chat.openai.com) with code summarisation tasks from real-world VB and COBOL programs and found that it was capable of providing appropriate explanations of small program extracts and functions from legacy code. The explanations were generally correct, although sometimes missing details.

The prompt used for VB6 code summarisation was:

"Can you explain the purpose of the following VB code, and the meaning of the function parameters?"

For COBOL, the prompt was:

"Can you explain the purpose of the following COBOL code and the role of each file?"

As an example of VB6 summarisation, we submitted after the VB prompt the following sample code extract from the utilities part of the industrial VB case:

```
FUNCTION Get_lr(M AS Variant,
    R_arr AS Range,
    C AS Variant) AS Double
 DIM N AS Long
 DIM I AS Integer

 N = R_arr.Rows.Count

 FOR I = 1 TO N - 1
   IF (M <= R_arr(I + 1, 2) AND
       M >= R_arr(I, 2)) THEN
     Get_lr = R_arr(I, C)
     EXIT FOR
   END IF
 NEXT I
END FUNCTION
```

The summary produced by GPT−3.5 for this function was:

"Overall, the function aims to perform a lookup operation within a specified range based on a given input value, returning the corresponding value from the specified column of the row where the input value falls within the range."

While this is correct, it misses some subtle points. For example, the code makes an implicit assumption that the column 2 values are in sorted ascending order, and GPT does not identify this condition. Likewise, there is an implicit assumption that C must be a valid column number of R_arr. A visual explanation could be preferable to a purely textual description in this case, because the $Range$ R_arr represents a rectangular area of cells on a spreadsheet.

We concluded from this investigation that an LLM with sufficient code and domain knowledge could be used to generate textual documentation for the abstracted system model. The generated summaries can be integrated into the semantic model as comments attached to model elements. However, as with any use of an LLM, the risk of erroneous and inconsistent output needs to be mitigated by validation of the output [56].

There are other forms of abstraction that could be useful, depending on the nature of the re-engineering task:

- Abstraction to an object-oriented specification. In this case a specific spreadsheet range could be represented instead as a list of instances of a class, with the columns of the range being the attributes of the class, and the rows being the objects.
- Mathematical specification in OCL. Loops of the above form could be abstracted to a postcondition such as:

$$post:\quad result = Sequence\{1, .., N\}{\rightarrow}any(I \mid Cond \cdot V)$$

where $Cond$ expresses the conditional test and V the returned value $R\_arr(I, C)$.

These forms of abstraction would require collaboration by both system experts and re-engineering experts (Fig. 1).

7 Code synthesis and test case generation

From the abstracted and restructured UML/OCL representation, code can be generated in multiple target languages. AgileUML provides code generators to synthesise code in ANSI C, C++, C#, Java, Go, Swift and Python. The generated code structure is closely aligned with the specification structure, facilitating traceability.

Together with target code, we also generate unit test cases for operations, and a mutation testing UI, to support test case selection for the synthesised code [30, 31]. Although generated code should be 'correct by construction', testing is also usually necessary in order to provide additional assurance to stakeholders in the case of critical system re-engineering. Testing will often be mandated by the customers, and can be the most time-consuming part of a re-engineering process [16].

We derive unit test cases for operations from the operation parameter types, and from the attribute types of the class to which the operation belongs. The tests are 'black box' tests and are independent of the operation functionality/activity. For each parameter or attribute of a given type T, a set of test values are chosen which represent typical values of T, together with potential boundary values. Thus for parameters or attributes of int type we use the values $0, 1, -1, 1024, -1025$ and $MAXINT$. The choice of test values for each type can be directly configured by the end user. If an operation has preconditions, then these are also used to infer and constrain the generated test values. For example, an operation:

```
operation integerPlaces(x : int) : int
pre: x > 0
post:
 result = 1 + x->log10()->oclAsType(int)
```

would have test values 0, 1, 1024 and $MAXINT$. Similarly, test values for attributes are adjusted based on any invariants of their class. All combinations of test values for parameters and attributes are used, so that an operation with N integer parameters could have more than 6^N test cases. This test generation procedure can therefore result in very large test sets, and we use mutation testing [6, 30, 32] to reduce these sets to a subset of effective tests.

Mutation testing is based on creating mutated versions of operations, with a single mutation in the code or postcondition definition of an operation *op* producing a mutant version *op*′. For the above example, a mutant version is:

```
operation integerPlaces_mut2(x \: int) : int
pre: x > 0
post:
 result = 1 + x->log()->oclAsType(int)
```

Tests for *op* are ranked based on the proportion of *op* mutants that they detect, i.e., the proportion of *op* mutants for which they give a different result compared to the original *op* version. For the above example, test $x = 1$ has 33% effectiveness in detecting mutants, whilst $x = 1024$ has 100% effectiveness. Other test values have 0% effectiveness. The higher the rate of detection, in principle the more effective is the test at discovering flaws in implementations of *op*. Tests which have low scores can be eliminated from the test set. Thus in the above case we would only retain test values 1 and 1024. The result is a reduced test set which can be used for unit testing and regression testing of the generated code for different target platforms.

In the case of specifications abstracted from COBOL, the variables are specified to have fixed data width, so test case generation can utilise this information to restrict the produced tests. For example, tests for a string attribute with size N would include random strings of length N, and a string consisting of N spaces.

8 Evaluation

In this section we provide evidence for research questions RQ1 and RQ2 in the context of VB6/VBA, COBOL '85 and Python re-engineering tasks.

8.1 RQ1: Effective re-engineering process

To answer this question we applied the AMDRE re-engineering process to three typical re-engineering tasks: (i) translation of VB6/VBA code to Python 3, with quality analysis, specification extraction, refactoring and rearchitecting of the abstracted code; (ii) translation of COBOL code to Java and C#, with quality analysis and refactoring of the abstracted code; (iii) abstraction of Python code to UML/OCL, with type inference and refactoring to improve code quality, prior to translation to Java.

As measures of translation accuracy we use two semantic accuracy measures:

1. The *computational accuracy* (CAcc) measure of [37]. This evaluates the percentage of tests which return the same results when applied to source and target versions of the re-engineered application. It is appropriate because users are generally concerned to preserve the functionality of the original version in the re-engineered system.
2. The *runtime equivalence* (REquiv) accuracy measure of [28]. This measures the percentage of programs which have 100% computational accuracy for their translation, i.e., all tests for the program return the same results in the original and translated versions.

In order to compute these measures we use the original test cases of example programs, and not the autogenerated tests produced by AgileUML from the abstracted semantic models (Sect. 7). The reason for this choice is that the original tests will be specific to the expected semantics of the programs, whilst the autogenerated tests are black-box tests independent of internal program semantics. To judge if the results of a test on the original and re-engineered codes are 'the same', we use a similarity function f_{sim} on the results. As with the corresponding function $f_{matched}$ of [28], f_{sim} considers numeric results to be the same based on their semantics, not their text (so that 39.5 and 3.95e+1 are considered the same output value, for example).

As measures of architectural quality we use the CRA-Index measure of [8], and the quality criteria ADP, ISP and DR of [49].

8.1.1 VB6/VBA re-engineering

Table 7 gives the accuracy results for translation of VB6/VBA source examples to Python 3.9. There are 100 VB6 examples, including 10 parts of the 2000 LOC bond analysis function suite. The tests for the parts of this case were provided by the business that owns the application. The other examples and tests are taken from [50]. We show the percentage of individual tests that have the same results for source and target versions (computational accuracy, CAcc), and also the percentage of cases which have equivalent behaviour for all their tests (runtime equivalence, REquiv). Grammar coverage of VB6 by the $\mathcal{CSTL}$ abstraction script is 320 of 412 gmar productions (77%).

Table 7 VB6 to Python evaluation cases: accuracy

Case category	Cases (tests)	Translation accuracy	
		CAcc (%)	REquiv (%)
Language	60 (123)	81	70
Functions	30 (74)	82	87
Bond app	10 (51)	90	60
Total	100 (248)	84	74

Table 8 VB6/VBA cases: reachitecting quality improvement

CRA: original	CRA: rearchitected	ADP	ISP	DR
0.18	0.97	100%	100%	80%

Table 8 shows the average CRA-Index before and after rearchitecting for 15 VB cases for which rearchitecting was applied. CRA-Index increased in 13 cases, was lower in 1 case and was unchanged in 1 case. The percentage of cases which satisfy ADP, ISP and DR after rearchitecting is also given.

For the industrial case, we performed analysis of the source code and abstracted UML/OCL, using an iterative process working together with the business representative to identify the domain semantics of individual parts of the system and to prioritise improvements which were considered to be of most value to the owning business.

It was found that the monolithic code of the application could be conceptually divided into two main parts: (i) computation of bond values and of the Nelson Siegel (NS) [52] and Nelson Siegel Svensson (NSS) yield curve models under varying assumptions—approximately 550 LOC; (ii) a genetic algorithm implementation of yield curve fitting [22], using a set of market data from a spreadsheet—approximately 1400 LOC. There are also small utility operations. Part (ii) makes use of part (i). Part (i) is well commented, but the more complex part (ii) has almost no comments, but instead 25 lines of commented-out code.

Some cases of EPL, EFO, EFI and DC were detected, but the most significant problems are:

- EFS: the main routine of part (ii) is over 500 LOC, and a secondary routine is over 300 LOC;
- MGN: 37 cases in part (i), primarily due to an embedded assumption that bond coupon frequency is twice per year, and 66 cases in part (ii), due to hard-wired assumptions about the structure of the associated spreadsheet, the number of yield curve parameters (6) and market data points (8), and other rigid assumptions which would make the code difficult to adapt.
- UVA: part (i) has 27 unused variables, apparently a legacy of a previous version of the code.

Table 9 VB6/VBA industrial case: quality improvement

	MGN	UVA	DC/DE	EPL	EFS	EFO
Original	103	27	15	12	2	2
Revised	0	0	8	4	0	0

Part (ii) uses direct access to Excel instead of via a DAO.

As restructuring actions, we factored the 2 excessively large operations into parts based on data dependencies, and replaced magic numbers by declared constants. We applied the 'extract value object' refactoring of Sect. 4 to part (i) to remove EPL cases.

There are four operations with a common group of five parameters $beta0$, $beta1$, $beta2$ and $tau1$, $tau2$:

- PV_NS(... 9 parameters...)
- PVD_NS(... 8 parameters...)
- PVF_NS(... 10 parameters...)
- NS_extended(... 6 parameters...)

There are also four operations with a common group of six parameters $beta0$, $beta1$, $beta2$, $beta3$ and $tau1$, $tau2$:

- PV_NSS(... 10 parameters...)
- PVD_NSS(... 9 parameters...)
- PVF_NSS(... 11 parameters...)
- NSS_extended(... 7 parameters...)

Applying 'extract value object' with the first group of common parameters resulted in a class NS with the 5 attributes corresponding to NS yield curve parameters, and applying the refactoring to the second group resulted in a class NSS with 6 attributes for the NSS parameters. Since the NS attributes are a subset of the NSS attributes, the NSS class can be expressed as a subclass of the NS class.

Further refactoring to remove DC and DE flaws involving multiple (> 2) clones was also carried out. UVA variables were removed. Furthermore, all the original code was contained in a single module. Analysis of the call graph confirmed that a hierarchical organisation implicitly existed, whereby part (ii) depended on part (i) and the utilities, and part (i) depended on the utilities. Using the heuristics of Sect. 4.3 the code was partitioned into classes for these implicit components, increasing component cohesion.

Table 9 shows the flaw counts of the case before and after restructuring.

Overall the total flaws were reduced from 161 to 12, and the flaw density reduced from 8 to 0.6%. Semantic preservation was satisfied for the deterministic numeric computations of part (i) and the utility programs. However, part (ii) involved calls of $Beta\_Inv$, $Norm\_Inv$ and Rnd, where the values produced by Excel/VB differ slightly from the translated versions of these functions.[6] In such cases the customer of the re-engineering project needs to decide if the differences are significant, and if so, an alternative implementation that simulates the source exactly would need to be created or selected.

8.1.2 COBOL '85 re-engineering

We examined 100 examples of COBOL code, including examples from the language manual [13] and textbook [57], together with the industrial case of [38]. Table 10 gives the accuracy results for translation of these cases to Java and C#. The tests were taken from the same sources as the examples. Grammar coverage of COBOL '85 is 456 of 631 gmar productions (72%). The main omissions from coverage are installation-specific aspects such as the data-base, report, program-library, communication and screen sections of a COBOL program.

GO TO statements are replaced by recursion during the abstraction process, and the 'replace recursion by iteration' transformation is then applied to remove this quality flaw.

An example of a poorly-structured COBOL program with a complex use of PERFORM is given in [57]:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. COBOLPERFORMSOURCE.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 CNT PIC 999 VALUE 0.
PROCEDURE DIVISION.
MAIN SECTION.
P1.
 MOVE ZERO TO CNT.
 PERFORM P2 THRU P5.
P2.
 PERFORM P3.
P3.
 ADD 1 TO CNT.
 IF CNT = 5
   MOVE ZERO TO CNT.
P4.
 ADD 3 TO CNT.
P5.
 DISPLAY CNT.
```

Although this is an artificial example, similar coding occurs in real-world cases such as the charge calculation program of [38]. The example program is abstracted to the UML/OCL specification:

```
class COBOLPERFORMSOURCE_Class {
 attribute CNT : int := 0;
 invariant 0 <= CNT & CNT <= 999;

 operation P1_Call()
```

[6] This arises because the actual function definitions used by Excel are not available.

Table 10 COBOL '85 to Java, C# evaluation cases: accuracy

Case category	Cases (tests)	Accuracy (Java)		Accuracy (C#)	
		CAcc (%)	REquiv (%)	CAcc (%)	REquiv (%)
Statements	65 (138)	93	80	94	91
Language	25 (58)	78	60	91	75
Functions	10 (33)	82	80	96	83
Total	100 (229)	87	75	94	86

```
pre: true post: true
activity: CNT := 0;
  self.P2_thru_P5() ;

operation P2_Call()
pre: true post: true
activity: self.P3_Call() ;

operation P3_Call()
pre: true post: true
activity:
  CNT := (CNT + 1)->truncateTo(0) ;
  if (CNT = 5) then (
    CNT := 0;
    skip) else skip ;

operation P4_Call()
pre: true post: true
activity:
  CNT := (CNT + 3)->truncateTo(0) ;

operation P5_Call()
pre: true post: true
activity:
  execute ("" + CNT)->display() ;

operation MAIN() pre: true
post: true
activity:
  CNT := 0;
  self.P2_thru_P5() ;
  self.P3_Call() ;
  CNT := (CNT + 1)->truncateTo(0) ;
  if (CNT = 5) then (
    CNT := 0;
    skip) else skip ;
  CNT := (CNT + 3)->truncateTo(0) ;
  execute ("" + CNT)->display() ;

  ...

operation P2_thru_P5() : void
pre: true post: true
activity:
  self.P2_Call() ;
  self.P3_thru_P5() ;

operation P3_thru_P5() : void
pre: true post: true
activity:
  self.P3_Call() ;
  self.P4_thru_P5() ;

operation P4_thru_P5() : void
 pre: true post: true
 activity:
   self.P4_Call() ;
   self.P5_Call() ;
}
```

This specification makes clear that the original program has a complex call graph structure:

$$MAIN \longrightarrow P2\_thru\_P5$$
$$MAIN \longrightarrow P3\_Call$$
$$P2\_thru\_P5 \longrightarrow P3\_thru\_P5$$
$$P2\_thru\_P5 \longrightarrow P2\_Call$$
$$P2\_Call \longrightarrow P3\_Call$$
$$P3\_thru\_P5 \longrightarrow P4\_thru\_P5$$
$$P3\_thru\_P5 \longrightarrow P3\_Call$$
$$P4\_thru\_P5 \longrightarrow P4\_Call$$
$$P4\_thru\_P5 \longrightarrow P5\_Call$$

By repeatedly applying the 'Replace call by definition' refactoring, the $MAIN$ operation code can be transformed to:

```
operation MAIN() : void
pre: true post: true
activity: CNT := 0  ;
  CNT := ( CNT + 1 )->truncateTo(0)  ;
  if ( CNT = 5 ) then
    ( CNT := 0  ; skip   )
  else skip  ;
  CNT := ( CNT + 1 )->truncateTo(0)  ;
  if ( CNT = 5 ) then
    ( CNT := 0  ; skip   )
  else skip  ;
  CNT := ( CNT + 3 )->truncateTo(0)  ;
  execute ( ( "" + CNT )->display() )    ;
  CNT := ( CNT + 1 )->truncateTo(0)  ;
  if ( CNT = 5 ) then
    ( CNT := 0  ; skip   )
  else skip  ;
  CNT := ( CNT + 1 )->truncateTo(0)  ;
  if ( CNT = 5 ) then
    ( CNT := 0  ; skip   )
  else skip  ;
  CNT := ( CNT + 3 )->truncateTo(0)  ;
  execute ( ( "" + CNT )->display() )  ;
```

This is then in a form which can be translated directly to Java.

Table 11 Python to Java evaluation cases: accuracy

Case category	Cases (tests)	Translation accuracy	
		CAcc (%)	REquiv (%)
Data structures	30 (72)	86	83
Statements	34 (74)	81	76
Libraries	9 (35)	89	67
Language	27 (45)	78	70
Total	100 (226)	83	76

Table 12 Python evaluation cases: restructuring

Flaw category	Original flaws	Flaws after restructuring
EDN	12	3
MDV	4	0
DE	4	0
PMV	2	2
EPL	3	1
Total	25	6

8.1.3 Python re-engineering

Table 11 shows the accuracy results for Python to Java translation. 100 Python 2 and Python 3 cases were considered. These figures can be compared to the best values of 78% for computational accuracy and 42% for runtime equivalence given for Python to Java translation in [28]. Grammar coverage of Python is 166 of 200 gmar productions (83%).

Examples of poorly-structured Python code were taken from undergraduate student machine learning programs. These included DE, EPL, EDN, PMV and MDV flaws. Table 12 shows the overall flaw counts before and after restructuring.

As an example of EDN, the following structure was present in one case of data pre-processing:

```
if cond1 :
 proc1
 if cond2 :
   proc2
   return e1
 else :
   proc3
   return e2
else :
 if cond3 :
   proc4
   return e3
 else :
   proc5
   return e4
```

This is an example of the 'pipeline jungle' antipattern, complex logic which is difficult to understand and maintain

Table 14 Example of effort required for re-engineering process steps

Step	Effort (person days)
Abstraction	0.5
Analysis	0.5
Restructuring	3
Specification extraction	3
Code synthesis	0.5
Test generation	0.5
Total	8

[63]. There are at least 8 statements at the maximum nesting depth of 3 (since this code is nested in a function definition).

By applying the 'Reduce code nesting' refactoring, this can be replaced by the functionally-equivalent code:

```
if cond1 then
 (proc1 ;
  if cond2 then
   (proc2;
    return e1) else skip ;
  proc3;
  return e2) else skip ;

if cond3 then (proc4;
 return e3) else skip;

proc5 ;
return e4
```

This version reorganises the pre-processing as a linear chain of filter elements. In this version there are only 2 effective statements at the maximum nesting depth. Other examples of Python re-engineering are given in the *avatarCases* directory of the provided dataset. These are typical Python solutions to programming problems, sourced from AtCoder [1].

8.2 RQ2: Re-engineering effort reduction

To answer this question we consider the typical processes involved in re-engineering, and identify to what extent our approach can automate or support these steps (Table 13).

This shows that AMDRE enables substantial automation of several main re-engineering steps, hence reducing the manual workload of these processes. Table 14 gives an illustration of the workload required for the AMDRE steps in an actual re-engineering case, the bond analysis application of Sect. 8.1.1. This was a relatively small legacy system, and it can be expected that timescales would be larger for more substantial systems, especially in the steps requiring detailed user intervention.

In addition, the abstraction and code-synthesis steps can be user-configured to address different source/target languages, to extend the source language coverage or to process installation-specific variations in languages. Each of the code abstractor scripts *VB2UML*, *COBOL2UML* and *Python2UML* are defined as $\mathcal{CSTL}$ text files external to the AgileUML toolset, and can be edited to modify or extend their processing, without specialised knowledge of the toolset. Only knowledge of the programming language grammar, of $\mathcal{CSTL}$, and of UML/OCL syntax is needed to perform such adaption.

For example, in order to add coverage of the recently-introduced 'map union' interpretation of the Python pipe operator to the $Python2UML$ script, we added two rules to the script:

```
_1 | _2 |-->_1->union(_2)<when> _1 Map
```

in the $expr$:: ruleset, and

```
|= _1 |-->->union(_1)<when> _1 Map
```

in the $assign\_part$:: ruleset.

Code-generators for the production of Java, Go and Swift code are also written as $\mathcal{CSTL}$ scripts, and can be adapted by end users independently of the AgileUML toolset.

It would also be possible for users to create new $\mathcal{CSTL}$ scripts to perform specific new analyses (e.g., to compute new metrics or to generate different forms of documentation) from source code parse trees. In general, a script $sc.cstl$ can be applied to a parse tree by the command line tool $cgtl$:

```
cgtl sc.cstl ast.txt
```

where $ast.txt$ contains the textual representation of the parse tree (as produced by an ANTLR parser).

The abstraction scripts are relatively small and required low resources to construct (Table 15). To further reduce effort, we provide a command-line tool $antlr2cstl$ to generate outline $\mathcal{CSTL}$ rulesets from ANTLR grammars. The outline versions of the COBOL rulesets shown in Sect. 3 were generated in this way from the COBOL grammar.

We have found that $\mathcal{CSTL}$ scripts are approximately 3 times faster to write than Java code, for the same reverse-engineering task. However, Java coding may be necessary for specialised tasks, and $\mathcal{CSTL}$ includes a facility for scripts to call external Java functions in such cases.

9 Threats to validity

Threats to validity include bias in the construction of the evaluation, inability to generalise the results, inappropriate constructs and inappropriate measures.

9.1 Threats to internal validity

9.1.1 Instrumental bias

This concerns the consistency of measures over the course of the analysis. To ensure consistency, all analysis and measurement was carried out in the same manner by a single individual (the first author) on all cases. Analysis and measurement for the results of Tables 7, 8, 9, 10, 11 and 12 were repeated in order to ensure the consistency of the results.

9.1.2 Selection bias

We chose VB6, COBOL '85 and Python examples which covered the core statements and features of the languages, and examples were taken from the language manuals [13] and [50] and from python.org. The large VB6 case is taken from a real-world application used in the second author's company, and exhibits characteristics typical of legacy financial VB applications. The large COBOL case is a real-world program from the utilities sector. Python programs included application code from student ML projects.

Table 13 Support for re-engineering processes

Step	Automation	Support
Abstraction	✓	Via ANTLR, $\mathcal{CSTL}$
Analysis	✓	Flaw measures; Guidance for flaw reduction
Restructuring	Partial	Manual selection of restructuring actions
Rearchitecting	Partial	Automation of heuristics for rearchitecting
Specification extraction	Partial	Interactive analysis; Code summarisation
Code synthesis	✓	AgileUML/$\mathcal{CSTL}$
Test generation	✓	Mutation testing

Table 15 CSTL script size and development effort

CSTL Script	Script LOC	Script effort (person months) (pm)
VB2UML	2430	4.5
COBOL2UML	3818	6
Python2UML	2246	3.5
Averages	2831	4.7

Table 16 Re-engineering approaches

Approach	Scope	Intermediate representation; tools	Methodology
Gra2MoL [27]	Model extraction	ASTM; ANTLR	Grammar-based
RCRRE [46]	Design extraction	SOFL	Pattern recognition
REMICS [35]	Modernisation; migration	KDM, UML	ADM + Scrum
SOAMIG [21]	Migration	TGraph; GReQL	SOMA
StateJ [64]	Design extraction	State machines	Symbolic execution
Transcoder [37]	Program translation	Implicit; Tensorflow	Machine learning
Tree2Tree [11]	Program translation	Implicit; PyTorch	Machine learning
XIRUP [20]	Modernisation; componentisa-tion	XSM custom metamodel; ATL	MDA; iterative
This paper (AMDRE)	Translation; Restructuring; Rearchitecting	UML/OCL; $\mathcal{CSTL}$, ANTLR	Agile & lightweight MDE

The tests used for checking semantic correctness were taken from the same sources as the examples, and were not constructed by the authors.

9.2 Threats to external validity: generalisation to different samples

Our approach is restricted to the analysis of business and data resource code, and does not directly handle GUI/UI code. Thus it is not applicable to code in the UI layer of applications. Here we have only considered the cases of mapping VB6 to Python, COBOL '85 to Java and C#, and Python to Java, however similar procedures have been followed to apply the approach for other 3GL language pairs, such as Pascal to Java translation, and the data for these are contained in the replication package.

A challenge with the re-engineering of legacy systems is that these are often closely bound to particular technologies such as Excel in VB finance cases, or particular data storage technologies and system services in the case of COBOL. These supporting technologies may themselves be legacy systems. Python ML systems may depend upon outdated versions of Python libraries and toolsets. In such cases the re-engineering process needs to consider the supporting software environment and dependencies of the legacy system, and decisions need to be made whether (i) use of the legacy environment should continue; (ii) the legacy environment should be simulated for the re-engineered system; or (iii) the re-engineered system should be transformed to depend instead upon modern platform services/libraries.

We facilitate the separation of the legacy system from its environment via the rearchitecting process (Sect. 4.3), which helps to isolate the platform dependencies of system components. In the case of Excel, we provide a library component that attempts to simulate Excel functionality. Likewise, we simulate some COBOL file storage mechanisms such as indexed-sequential files, using the OclFile library of AgileUML. However, in general this issue may require significant additional work to resolve, which is not addressed by the AMDRE process.

9.3 Threats to construct validity: inexact characterisation of constructs

We have used established metrics such as the CRA-Index [8] and computational accuracy [37] measures in our evaluation, in order to precisely measure semantic preservation and structural improvement resulting from the re-engineering process. In addition, the principles ADP, ISP and DR from the clean architecture [49] have been used as criteria for the quality of the re-engineered system.

9.4 Threats to content validity

9.4.1 Relevance

The re-engineering approach has been shown to be applicable to the analysis and translation of the main COBOL '85, VB6 and Python program elements. Here we have focussed on specific target languages (Java, C# and Python), but other target languages could also be used, such as Swift and C++.

9.4.2 Representativeness

The industrial finance application was selected to be representative of legacy systems at the company, a finance SME. The combination of Excel code and VB code in this application is typical of VBA programs in the finance domain. The large COBOL application is typical of legacy business application code. The Python examples are representative of Python ML system coding.

Table 17 Comparison of re-engineering effectiveness

Approach	CAcc (%)	REquiv (%)
Transcoder	56.1	2.0
This paper	83	76

9.5 Threats to conclusion validity

The preservation of program semantics via abstraction and restructuring is a key element of our process. Semantic preservation cannot be assured in all cases because differences exist between different program execution environments due to different device characteristics and operating system behaviour. The semantics of libraries used by programs may not be available, so that no assurance about semantic preservation can be made unless exactly the same library in the same environment is also used in the re-engineered system. A particular case is Excel, for which the WorksheetFunction library can be used in VBA code, but the exact definition of the library functions is not publicly available. Therefore there may be minor discrepancies between the Excel definition of functions such as the cumulative normal distribution, and our version of these functions in the OCL Excel library component.

10 Related work

The most direct form of re-engineering is program-to-program translation, where normally no abstractions above the level of syntax trees are constructed. This is an active area of current research [23, 45, 53]. Both machine learning approaches and explicit rule-based translations have been used for program translation [11, 28, 29, 37]. Direct translation has the disadvantage that no abstractions or documentation of the code are produced, and that N*M separate translations need to be defined or learned for N source and M target languages. Replicating the structures and idioms of a legacy system in a new language may result in code that is difficult to understand and maintain without knowledge of the legacy code concepts [16].

Other re-engineering approaches attempt to recover some form of design or specification from the code, in order to support the redevelopment and improvement of the system [58, 59]. Inevitably some manual intervention is needed in this process, as specialised domain expertise is necessary to interpret the program data and functionality in domain terms. For example, in the case of the bond analysis system given above, the domain expert was able to identify that variables B1 and B2 (Sect. 2.1) represent the Nelson-Siegel yield curve model parameters β_0 and β_1 [52].

Redesign and transformation of a system specification is necessary if the existing system has poor quality or if the target language/platform is substantially different to the source. This may involve transforming procedural program designs into object-oriented or declarative designs [38, 46]. However, such a radical rewrite also carries the risk of loss of essential information from the source, and can make it more difficult to understand for maintainers familiar with the old system.

Most model-driven re-engineering (MDRE) approaches have followed the ADM approach [35, 59, 60]. Whilst agile MDE is an active area of research [3] there have been few works applying agile MDE to re-engineering. The closest work to ours is [24], but they focus on the business level organisation and prioritisation of re-engineering instead of the technical process.

In previous work we have described $\mathcal{CSTL}$ and its application to program translation [40, 41, 44]. In this paper we consider in more depth the abstraction processes and semantic representations for COBOL, VB and Python, and we extend the program translation process to a general MDRE process including refactoring and rearchitecting of applications based on their abstracted semantic models.

Table 16 summarises recent related approaches and how they compare to our approach. Table 17 compares the effectiveness of our approach for Python to Java re-engineering with the Transcoder ML approach of [37]. This is the only approach for which comparable data was available. Transcoder has been regarded as the state-of-the-art approach for ML-based transpilation [28]. The results show that our approach can achieve higher accuracy than Transcoder on similar program translation tasks. In addition, compared to Transcoder, our approach additionally enables the abstraction of designs and specifications from code, and the refactoring of designs for quality improvement.

Overall our approach is distinguished from previous MDRE approaches by (i) construction of a *semantic* intermediate representation in a standard software language, instead of a repository of syntactic data in a metamodel; (ii) providing rearchitecting facilities linked to ‘clean architecture’ qualities; (iii) providing user-configurable tools that do not require high skills in MDE.

11 Conclusions and future work

In this paper we have defined the AMDRE approach for agile model-driven re-engineering, which provides restructuring and rearchitecting processes for legacy system quality improvement and platform migration, based upon detailed semantic models abstracted from code. We evaluated the approach on real-world legacy systems in VB, COBOL and Python, and demonstrated that AMDRE can be applied to practical cases of re-engineering tasks.

The use of model-driven techniques for software re-engineering supports automation of the source code abstraction process, and the definition of a wide range of analyses upon the abstracted models, and facilitates the production of target code in multiple languages/platforms. It also enables strong assurance of semantic preservation to be made, which is of key importance to industrial clients.

In future work we will investigate the automated learning of abstraction transformations from examples, using symbolic machine learning [43]. This would enable users to specify code abstractors without the need to write specifications in $\mathcal{CSTL}$. We will also further investigate the use of LLMs for specification discovery, including the creation of specialised LLMs for this task by the fine-turning of general LLMs [65, 72]. An increasingly important area of research is *software sustainabiity*, the reduction of software energy use and consequent greenhouse gas emissions [51]. Our approach could also potentially be used to analyse energy use flaws (energy-inefficient practices in code) and to restructure the system to remove such flaws, analogously to the use of refactoring to remove quality flaws.

Acknowledgements Hessa Alfraihi is funded by Princess Nourah bint Abdulrahman University Researchers Supporting Project Number (PNURSP2024R411), Princess Nourah bint Abdulrahman University, Riyadh, Saudi Arabia. The support of Holistic Risk Solutions Ltd and CLMS UK Ltd is acknowledged. Kunxiang Jin carried out the research into mutation testing. Hanan Siala contributed to the background research on related work.

Data availability The AMDRE tools and evaluation data are at: zenodo.org/records/11073481.

Declarations

Conflict of interest The authors confirm that there are no conflict of interest.

References

1. Ahmad W, Tushar M, Chakraborty S, Chang K-W (2023) AVATAR: a parallel corpus for Java-Python program translation, arXiv:2108.11590v2
2. Agarwal M, Talamadupula K, Martinez F, Houde S, Muller M, Richards J, Ross S, Weisz J (2021) Using document similarity methods to create parallel datasets for code translation, arXiv:2110.05423v1
3. Alfraihi H, Lano K (2017) The integration of agile development and MDE: a systematic literature review, Modelsward
4. Beck K et al (2001) Manifesto for Agile Software Development, agilemanifesto.org
5. Behutiye W, Rodriquez P, Oivo M, Tosun A (2017) Analysing the concept of technical debt in the context of agile software development. IST 82:139–158
6. Belli F et al (2016) Model-based mutation testing—approach and case studies. SCP 120:25–48
7. Bowen JP, Breuer P, Lano K (1993) Formal specifications in software maintenance: from code to Z++ and back again. Inf Softw Technol 35(11–12):679–690
8. Bowman M, Briand L, Labiche Y (2010) Solving the class-responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. IEEE TSE 36(6):817–837
9. Buttner F, Gogolla M (2014) On OCL-based imperative languages. Sci Comput Program 92:162–178
10. Campbell G, Papapetrou P (2013) SonarQube in Action. Manning Publications Co
11. Chen X, Liu C, Song D (2018) Tree-to-tree neural networks for program translation. NIPS
12. Ciesielski K (1997) Set theory for the working mathematician. Cambridge University Press, Cambridge
13. ClearPath Enterprise Servers, COBOL ANSI-85 Programming Reference Manual, April 2015
14. Crawford M (1990) Lurking within COBOL PERFORMs. J Softw Maint 2(1):33–60
15. Deltombe G, Le Goaer O, Barbier F (2012) Bridging KDM and ASTM for model-driven software modernization, SEKE
16. De Marco A, Iancu V, Asinofsky I (2018) COBOL to Java and newspapers still get delivered. In: Proceedings IEEE international conference on software maintenance and evolution. IEEE Press, pp 583–586
17. da S Maldonado E, Shihab E, Tsantalis N (2017) Using natural language processing to automatically detect self-admitted technical debt, IEEE TSE. https://doi.org/10.1109/TSE.2017.2654244
18. Eclipse AgileUML project. https://projects.eclipse.org/projects/modeling.agileuml, 2024
19. Fowler M, Beck K (2019) Refactoring: improving the design of existing code, 2nd edn. Pearson, London
20. Fuentes-Fernandez R, Pavon J, Garijo F (2012) A model-driven process for the modernisation of component-based systems. Sci Comput Program 77:247–269
21. Fuhr A, Horn T, Riediger V, Winter A (2013) Model-driven software migration into service-oriented architectures. Comput Sci Res Dev 28:35–84
22. Gimeno R, Nave J (2006) Genetic algorithm estimation of interest rate term structure, Banco de Espana, report 0634

23. Guo D et al (2021) GraphCodeBERT: pre-training code representations with dataflow. In: ICLR 2021
24. Hadar E, Connelly K, Lagunova O (2009) Agile evolution of information systems using MDA of IT services. In: Proceedings of architecture in an agile world, vol 25
25. He X, Avgeriou P, Liang P, Li Z (2016) Technical debt in MDE: a case study on GMF/EMF-based projects, MODELS
26. IBM (2023) PL/I. www.ibm.com/products/pli-compiler-zos
27. Izquierdo J, Molina J (2014) Extracting models from source code in software modernisation. SoSyM 13:713–734
28. Jana P et al (2023) Attention, compilation, and solver-based symbolic analysis are all you need, arXiv:2306.06755v1
29. JavaScripthon (2022) https://extendsclass.com/python-to-javascript.htmln
30. Jin K, Lano K (2022) Design and classification of mutation operators for OCL specification. In: OCL 2022, MODELS
31. Jin K, Lano K (2022) OCL-based test case prioritisation using AgileUML, ModeVVa 2022, MODELS
32. Just R et al (2014) Are mutants a valid substitute for real faults in software testing? Proceedings of the 22nd ACM SIGSOFT ISFSE, pp 654–665
33. Kerner S (2023) COBOL language still in demand as application modernization efforts take hold. www.itprotoday.com
34. Khadka R et al (2014) How do professionals perceive legacy systems and software modernization? In: ICSE 2014. ACM Press
35. Krasteva I, Stavru S, Ilieva S (2013) Agile software modernization to the service cloud. In: ICIW, pp 1–9
36. Kurtz T (1978) BASIC, ACM history of programming languages conference. In: SIGPLAN notices, vol 13, no 8, pp 103–118
37. Lachaux M-A, Roziere B, Chanussot L, Lample G (2020) Unsupervised translation of programming languages, arXiv:2006.03511v3
38. Lano K, Malik N (1999) Mapping procedural patterns to object-oriented design patterns. Autom Softw Eng 6(3):265–289
39. Lano K (2017) Agile model-based development using UML-RSDS. CRC Press, Boca Raton
40. Lano K, Xue Q, Kolahdouz-Rahimi S (2020) Agile specification of code generators for model-driven engineering. In: ICSEA
41. Lano K (2022) Program translation using model-driven engineering, short paper. In: ICSE
42. Lano K, Kolahdouz-Rahimi S, Jin K (2022) OCL libraries for software specification and representation. In: OCL 2022, MODELS
43. Lano K, Xue Q (2023) Code generation by example using symbolic machine learning. Springer Nature Computer Science. Springer, Berlin
44. Lano K, Siala H (2024) Using MDE to automate software language translation. Autom Softw Eng 31:66
45. Le TH, Chen H, Babar M (2020) Deep learning for source code modeling and generation, arXiv:2002.05442v1
46. Liu S, Li H, Jiang Z, Li X, Liu F, Zhong Y (2021) Rigorous code review by reverse engineering. Inf Softw Technol 133:66
47. Liu X, Yang H, Zedan H (1997) Formal methods for the re-engineering of computing systems. In: Compsac'97
48. Marinescu R (2012) Assessing technical debt by identifying design flaws in software systems. IBM J Res Dev 56(5):66
49. Martin R (2018) Clean architecture. Prentice Hall, Englewood Cliffs
50. Microsoft Com (2022) Office VBA Reference. https://learn.microsoft.com/en-us/office/vba/api/overview
51. Nabavi E et al (2019) AI for sustainability: a changing landscape. In: Artificial intelligence for better or worse, future leaders
52. Nelson C, Siegel A (1987) Parsimonious modelling of yield curves. J Bus 60(4):473–489
53. Nguyen A, Nguyen T, Nguyen T (2013) Lexical statistical machine translation for language migration. In: 9th Joint meeting on foundations of software engineering, pp 651–654
54. Ogheneovo E (2014) On the relationship between software complexity and maintenance costs. J Comput Commun 2:1–16
55. OMG (2014) Object Constraint Language 2.4 Specification, OMG document formal/2014-02-03
56. Ouyang S, Zhang J, Harman M, Wang M (2023) LLM is like a box of chocolates: the non-determinism of ChatGPT in code generation, arXiv:2308.02828v1
57. Parkin A (1982) COBOL for students. Edward Arnold, London
58. Perez J et al (2003) Data reverse engineering of legacy databases to OO conceptual schemas. ENTCS 72(4):7–19
59. Perez-Castillo R, Garcia-Rodriguez de Guzman I, Piattini M (2010) Implementing business process recovery patterns through QVT transformations. In: ICMT
60. Perez-Castillo R, Garcia-Rodriguez de Guzman I, Piattini M (2011) Knowledge discovery metamodel ISO/IEC 19506: a standard to modernize legacy systems. Comput Stand Interfaces 33:519–532
61. Sammet J (1978) The early history of COBOL. In: ACM history of programming languages conference, SIGPLAN notices, vol 13(no 8), pp 121–161
62. Sanders J (2019) https://www.techrepublic.com/article/jpmorgans-athena-has-35-million-lines-of-python-code-and-wont-be-updated-to-python-3-in-time
63. Sculley D et al (2015) Hidden technical debt in machine learning systems. In: NIPS'15
64. Sen T, Mall R (2016) Extracting finite-state representation of Java programs. SoSyM 15(2):497–511
65. Siala H (2024) Enhancing model-driven reverse engineering using machine learning. In: ICSE'24 doctoral symposium
66. Sneed H, Jandrasics G (1987) Inverse transformation of software from code to specification. IEEE conference on soft maintenance
67. Sneed H (2011) Migrating from COBOL to Java: a report from the field. In: IEEE Proceedings of the 26th ICSM. IEEE Press, pp 1–7
68. Stavru S, Krasteva I, Ilieva S (2013) Challenges of model-driven modernization: an agile perspective, Modelsward
69. Tornhill A, Borg M (2022) Code Red: the business impact of code quality, arXiv:2203.04374v1
70. Wehaibi S, Shihab E, Guerrouj L (2016) Examining the impact of self-admitted technical debt on software quality. In: 23rd IEEE international conference on software analysis, evolution and re-engineering
71. Zhang C et al (2022) A survey of automatic source code summarization. In: Symmetry, vol 14, MDPI
72. Zhao W et al (2023) A survey of large language models, arXiv:2303.18223v10