Program abstraction by transformation: Abstraction of Visual Basic to UML

Kevin Lano[1], S. Kolahdouz-Rahimi[2]

[1]King's College London, Strand, London, UK
[2]University of Roehampton, Roehampton, London, UK

Abstract
Program abstraction is a key step in the extraction of information from executable code, in order to understand legacy code, produce documentation in the form of models, or to perform re-engineering to an alternative program platform/language. Several special-purpose model transformation languages have been developed to perform program abstraction, however it remains an open research question what kinds of transformation facilities and techniques are most appropriate for the problem. In this case, we define a task for abstracting a subset of VB6/VBA to UML and OCL, this task can be used to perform comparative evaluation of different transformation approaches for the abstraction problem.

Keywords
Program abstraction, Model-driven engineering, Reverse-engineering, Re-engineering

1. Introduction

Program abstraction is the process of extracting formalised information from executable program code. The input could be either source code [14] or object code [15], and the outputs can include data flow or control flow information. The purpose could be for program comprehension [5] or for refactoring or other quality improvement of the source [4]. Here we will focus on the task of abstracting software models from source code, for the purpose of re-engineering, in particular for translating the source code to a different programming language [9].

An important property in this situation is *semantic preservation*: the abstraction should accurately capture the semantics of the source code, in order that the organisation which owns the code can have confidence that the re-engineered version still performs the same functionality as the original. Thus the abstraction needs to be expressed in a language which supports detailed specification of behaviour. Here we propose the use of OCL [13, 1] together with UML class specifications, however other appropriate formalisms, such as activity diagrams or state machines, could also be used. The advantage of UML/OCL is that this is a widely-used specification formalism, with established semantics and a large number of tools available for analysis and for forward engineering to diverse target languages.

Program abstraction involves the co-use of parsing/grammar-based technologies with model-based technologies such as transformations. This is a similar situation to the combined use of grammar-based and model-based techniques for DSL tooling [2].

The specific re-engineering task is translation from Visual Basic version 6 (VB6) to Python version 3.9. To make the task practical, only a small subset of VB6 will be considered here, essentially modules with a top-level linear sequence of variable declarations and assignments.

The research questions we wish to investigate are:

RQ1 What form of transformation language and transformation language facilities are particularly effective for program abstraction?

RQ2 What are the specific challenges of defining program abstraction transformations?

RQ3 Are there any transformation design patterns or idioms which are particularly relevant for this domain?

RQ4 How should parsing and grammar-based technologies be integrated with transformations for program abstraction?

The case materials are available at: https://zenodo.org/records/7801436.

2. Visual BASIC

BASIC[1] was intended, as its name suggests, as a language for inexperienced programmers to use for relatively simple programming problems. It became popular with the

[1]Beginner's All Purpose Symbolic Instruction Code

TTC'23: 15th Transformation Tool Contest, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. A. Boronat, A. García-Domínguez, and G. Hinkel, 20 July 2023, Leicester, UK.
kevin.lano@kcl.ac.uk (K. Lano); shekoufeh.rahimi@roehampton.ac.uk (S. Kolahdouz-Rahimi)
0000-0002-9706-1410 (K. Lano)

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 MS applications such as Excel [12]. Visual Basic 6.0 (VB6), released in 1998, was the last version of VB prior to VB .NET, and is still supported on Windows platforms.

The principal 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 kinds of statements (67 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.

To make the case practical for TTC, we restrict the considered subset of VB6 to those programs written using variable declarations (DIM statements), assignment statements (including LSET, RSET and REDIM) and sequencing. We use the ANTLR version 4 VB6 grammar VisualBasic6.g4, which is supplied with the case materials, together with the executable Java parser generated from the grammar.

In terms of this grammar, the case concerns programs parsed according to the grammar/lexical rules for *module*, *moduleBody*, *moduleBodyElement*, *moduleBlock*, *block*, *blockStmt*, *letStmt* (restricted to the form *LHS* = *RHS*), *variableStmt*, *implicitCallStmt_InStmt*, *valueStmt*, *variableListStmt*, *variableSubStmt*, *subscripts*, *asTypeClause*, *iCS_S_VariableOrProcedureCall*, *iCS_S_ProcedureOrArrayCall*, *ambiguousIdentifier*, *baseType*, *complexType*, *argsCall*, *argCall*, *type_*, *subscript_*, *IDENTIFIER*, *literal*, *doubleLiteral*, *integerLiteral*, *STRINGLITERAL*, *TRUE*, *FALSE*, *rsetStmt*, *lsetStmt*, *redimStmt*, *redimSubStmt*, *iCS_B_MemberProcedureCall*, *iCS_S_MemberCall*. The VB6 grammar *VisualBasic*6.*g*4 is included in the case materials *grammar* directory.

Figure 1 shows the metamodel of the considered subset of VB6. This is available as an EMF metamodel in the case materials. LSET and RSET statements are combined with LET statements in this metamodel.

For example, the statement

```
X(2) = Y
```

would be expressed in terms of this metamodel as an instance

$$s : letStmt$$

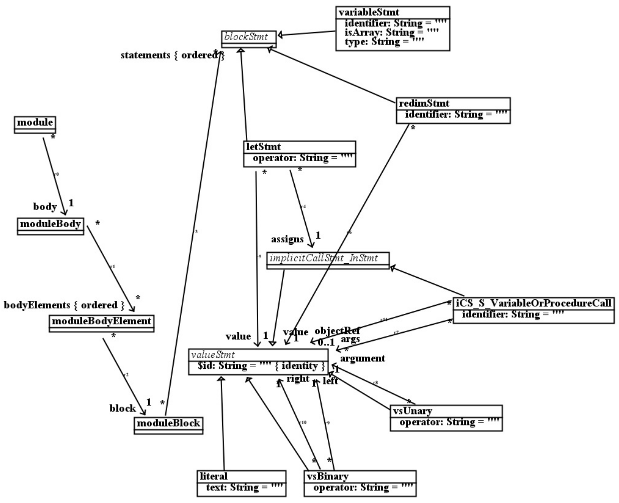

Figure 1: VB6 subset metamodel

where

$$\begin{aligned}
&s.operator = \text{“} = \text{”}\\
&s.assigns = lhs\\
&s.value = rhs\\
&lhs : iCS\_S\_VariableOrProcedureCall\\
&lhs.identifier = \text{“}X\text{”}\\
&lit2 : literal\\
&lit2 : lhs.args\\
&lit2.text = \text{“}2\text{”}\\
&rhs : iCS\_S\_VariableOrProcedureCall\\
&rhs.identifier = \text{“}Y\text{”}
\end{aligned}$$

3. Case Specification

The intended mapping from the VB6 subset to UML/OCL is as follows. The mapping of types is shown in Table 1. t' denotes the translation of t.

Table 1
Mapping of VB6 types to OCL types

VB6 type t	*UML/OCL translation t′*
Boolean, *Integer*	*Boolean*, *Integer*
Long, *LongLong*	*Integer*, *Integer*
String	*String*
Float, *Double*	*Real*
Array type $t()$	$Sequence(t')$
Collection	$Sequence(Map(String, OclAny))$

The VB6 data types Boolean, Integer (16-bit integers), Long (32-bit integers) and String translate directly to OCL types. However the VB6 Double is a semantically distinct subset (IEEE 754 64-bit floating point range) of OCL Real.

A specific computational type *double* with the necessary properties could be used to abstract VB6 Double. The VB6 Collection type is conceptually an ordered map type, whereby elements can be accessed by index as well as by key. One way to model an ordered map in OCL is as a sequence of individual maplets (single-element maps). Collection operators are then translated as in Tables 3, 4.

For each basic VB6 type t, there is a default OCL value $default_t$ for the type: 0 for integer types, 0.0 for floating-point types, *false* for booleans, and the empty string “” for strings.

The mapping of expressions is shown in Tables 2 and 3, where e' denotes the translation of e.

Table 2
Mapping of VB6 expressions to OCL expressions

VB6 source expression e	*UML/OCL translation e′*
Numeric literal v	v
String literal `"s"`	`"s"`
True, *False*	*true*, *false*
Nothing	*null*
Identifier *id*	*id*
Array access $e(v)$	$e' \rightarrow at(v')$
Bracketed expression (e)	(e')
Floor(x)	$(x').floor()$
Max(s)	$Set\{s'\} \rightarrow max()$
Min(s)	$Set\{s'\} \rightarrow min()$
Pow(x,y)	$(x').pow(y')$
Len(s)	$(s') \rightarrow size()$
Mid(s,i,j)	$(s').substring(i', i' + j' - 1)$
Unary expressions	
$+e$, $-e$	e', $-e'$
NOT(e)	$not(e')$
Binary expressions	
$e1 + e2$, $e1 - e2$,	$e1' + e2'$, $e1' - e2'$,
$e1 * e2$, $e1/e2$,	$e1' * e2'$, $e1'/e2'$,
$e1 \backslash e2$, $e1 \frown e2$,	$e1'$ *div* $e2'$, $(e1').pow(e2')$,
$e1 < e2$, $e1 <= e2$,	$e1' < e2'$, $e1' <= e2'$,
$e1 <> e2$, $e1 = e2$,	$e1' \; /= \; e2'$, $e1' = e2'$,
$e1 > e2$, $e1 >= e2$,	$e1' > e2'$, $e1' >= e2'$,
$e1$ & $e2$ String $e1$	$e1' + e2'$
$e1$ & $e2$ integer $e1$	$MathLib.bitwiseAnd(e1', e2')$
$e1$ *MOD* $e2$	$e1'$ *mod* $e2'$
$e1$ *AND* $e2$, $e1$ *OR* $e2$	$e1'$ *and* $e2'$, $e1'$ *or* $e2'$
$e1$ *XOR* $e2$	$e1'$ *xor* $e2'$
$e1$ *IMP* $e2$, $e1$ *EQV* $e2$	$e1'$ *implies* $e2'$, $(e1' = e2')$
$e1$ *LIKE* $e2$	$(e1') \rightarrow isMatch(e2')$

Table 4 describes the mapping of VB6 statements to procedural OCL. A VB6 module is mapped to a UML class with an operation for the module body.

4. Solution Criteria

The case tasks are:

Table 3
Mapping of VB6 collection expressions to OCL expressions

VB6 expression e	*UML/OCL translation e′*
NEW Collection	$Sequence\{\}$
$id.Item(v)$	$id \rightarrow select(m \mid$
$id(v)$	$m \rightarrow keys() \rightarrow includes(v')) \rightarrow any() \rightarrow at(v')$
id . Count	$id \rightarrow size()$
id . Items	$id \rightarrow collect(m \mid m \rightarrow values() \rightarrow any())$
id . Keys	$id \rightarrow collect(m \mid m \rightarrow keys() \rightarrow any())$
id . RemoveAll	$Sequence\{\}$

Abstraction: Implement the specified mapping from the VB6 subset to UML/OCL, using your chosen parsing technology and transformation language/languages.

Validation: Check that the test cases are correctly abstracted, by inspection or by execution/testing of the abstracted specification.

Translation (optional): translate the abstracted UML/OCL into Python 3.9 and test that the result satisfies the expected semantics. An existing MDE toolset or code generator can be used for this part.

The specific criteria to be evaluated are:

Coverage and completeness: the abstraction transformation should be able to process the given 10 example programs. This includes the abstraction of the VB6 types *Double*, *Integer*, *Long*, *Boolean*, *String*, *Collection* to appropriate UML/OCL types.

Correctness: the abstracted specifications should be correct with respect to the mapping of Tables 1, 2, 3, 4.

Efficiency: the abstraction process should be of practical efficiency (i.e., execution time less than 15 seconds for examples of 500 LOC, and a linear time complexity).

10 small VB6 examples are provided in the *examples* directory, both in source code form and as parse trees generated by the ANTLR VB6 parser. Your solution should correctly abstract these examples and optionally translate them to correct Python code. Compute the percentage of cases which are correctly abstracted/translated.

Test cases for each program are specified in the directory *tests*. There are 21 test cases in total. Compute the overall percentage of test cases which have the same result as the source in (1) their UML/OCL representation; (2, optional) the Python target code.

Five larger examples for testing performance are given in the *performance* directory. Compute the execution

Table 4
Mapping of VB6 statements to OCL statements

VB6 statement s	*UML/OCL translation s′*
DIM *id* AS *t*	var $id : t' := default_t$
DIM *id*	var $id : OclAny := null$
DIM *id*() AS *t*	var $id : Sequence(t') := Sequence\{default_t\}$
DIM *id*()	var $id : Sequence(OclAny) := Sequence\{null\}$
$id = e$	$id := e'$
$e(v) = value$	$e' := e'.setAt(v', value')$
id . Add *v*	$id := id{\rightarrow}including(Map\{null \mapsto v'\})$
id . Add Key := *k*, Item := *v*	$id := id{\rightarrow}including(Map\{k' \mapsto v'\})$
id . Remove *v*	$id := id{\rightarrow}excludingAt(v')$ when *v* integer
id . Remove *v*	$id := id{\rightarrow}select(m \mid m{\rightarrow}keys(){\rightarrow}excludes(v'))$ when *v* string
LSET $id = e$	$id := StringLib.leftAlignInto(e', id.size)$
REDIM $id(val)$	$id := Sequence\{1..(val')\}{\rightarrow}collect(id{\rightarrow}any())$
RSET $id = e2$	$id := StringLib.rightAlignInto(e', id.size)$

time of your approach on these examples, as an average of three executions. Also provide a specification of your execution environment.

Desirable characteristics of solutions are (1) clear and modular expression of abstraction rules, for example, that the abstraction of each source language construct is defined by a specific transformation rule for that construct; (2) efficient processing of program source data and generation of target text; (3) preservation of source code structure in the abstraction and target, in order to enhance traceability; (4) adaptable and extensible transformations, which could be extended to process larger subsets of VB using the same transformation approach.

Note that whitespace and new lines are sometimes significant for VB elements, such as the format of functions and subroutines. Thus the ANTLR grammar explicitly refers to these lexical elements in the grammar rules. They can be removed from ASTs prior to processing by an abstraction transformation, however.

4.1. Scores for solutions

Solutions will be evaluated according to these measures:

1. *Corr*1: The percentage of the 10 example programs which are correctly abstracted to UML/OCL (also optionally: the percentage correctly translated to Python)
2. *Corr*2: The percentage of the 21 tests which have the same result in the source and abstraction (also optionally: in the source and the translation to Python)
3. *Perf*: Percentage of performance examples for which your approach has the same or better performance than the reference solution, on similar hardware.

5. Journal Publication

Case solutions which meet a threshold standard of capabilities and scores will be selected for incorporation into a JOT article. JOT is an appropriate venue as it is concerned with the application of MDE technologies in practical software development contexts. Re-engineering of legacy systems into modernised and object-oriented platforms/languages is of high concern to businesses that utilise software [7]. Although no solutions were submitted before TTC 2023, we have subsequently issued calls for solutions on relevant forums such as Strumenta.

6. Reference Solution

A solution to the abstraction part of the case is provided in the *solution* directory, using the CGTL/CSTL text-to-text transformation language [8, 10]. The *cgtl* command-line tool for executing CGTL scripts can be obtained from the AgileUML Github repository[2] or from agilemde.co.uk. We also provide it in the Zenodo repository of this paper.

The VB2UML.cstl script, together with vbDeclarations.cstl and vbFunctions.cstl, defines abstraction rules for the VB6 grammar. The scripts cover 77% of the VisualBasic6.g4 grammar productions including the subset considered in this case. As discussed above, whitespace lexical items from the grammar are discarded before being processed by the scripts.

For example, the VB6 grammar definition for the *valueStmt* non-terminal includes the BNF productions with 36 implicit syntactic cases:

```
valueStmt:
  ...
```

[2]github.com/eclipse/agileuml

```
  | valueStmt WS? AMPERSAND WS? valueStmt
  | valueStmt WS? (EQ | NEQ |
          LT | GT | LEQ |
          GEQ | LIKE | IS) WS? valueStmt
```

Note that ANTLR uses a top down and left-to-right prioritisation of grammar productions within a grammar rule. Thus `V & W = Z` is parsed as $(valueStmt\ (valueStmt\ V\ \&\ W) = Z)$.

The corresponding abstraction ruleset *valueStmt*:: therefore has a rule for each of the 9 binary operators of these 36 cases (the rules ignore whitespace occurrences):

```
valueStmt::
...
_1 & _2 |-->(_1 + _2)<when> _1 String
_1 & _2 |-->MathLib.bitwiseAnd(_1,_2)
_1 = _2 |-->_1 = _2
_1 <> _2 |-->_1 /= _2
_1 < _2 |-->_1 < _2
_1 > _2 |-->_1 > _2
_1 <= _2 |-->_1 <= _2
_1 >= _2 |-->_1 >= _2
_1 LIKE _2 |-->(_1)->isMatch(_2)
_1 IS _2 |-->_1 <>= _2
```

Likewise for other forms of expression and statement. A CGTL/CSTL rule

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

of ruleset *tg*:: matches against AST terms with tag *tg* which correspond element-by-element to the LHS tokens. E.g., a term *t* of form $(valueStmt\ t1\ \&\ t2)$ will match against the LHS of the *valueStmt* rule

```
_1 & _2 |-->(_1 + _2)<when> _1 String
```

with *t*1 bound to _1 and *t*2 bound to _2. If the condition also holds, then the rule is selected for application. Upon application, the input subterms *t*1 and *t*2 are recursively mapped to strings *s*1 and *s*2, and the result of the rule formed as the substitution `RHS[s1/_1, s2/_2]`, in this case this is `(s1 + s2)`.

User-defined functions *f* can also be applied to terms by the notation *_i*'*f*, where *f* is defined by a ruleset *f*::. This enables processing of subterms of a term bound to *_i*. Source terms can be inspected to any depth using this technique. More details of the reference solution are given in [11].

For forward engineering to Python, the Python code generator of AgileUML[3] is used.

Figure 2 shows the overall execution time for abstraction of the five performance examples. The time is computed as the average of 3 executions, on a Windows 10 quad-core laptop (Intel i5-7440HQ 2.8GHz processor, 8GB RAM) with 25% processor allocation.

[3]github.com/eclipse/agileuml

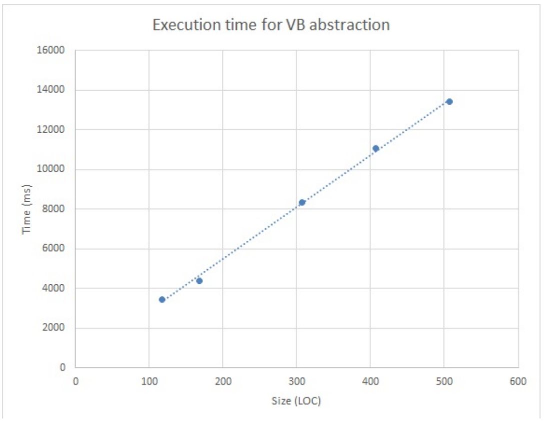

Figure 2: Performance of reference solution

With regard to the research questions of Section 1, the reference solution has the following characteristics:

RQ1: Transformation language/facilities CGTL is designed for the processing of parse trees produced by a language grammar/parser. This has the advantage that there is no need to define and populate a metamodel, but the processing capabilities are also more restricted compared to a general model transformation language.

RQ2: Challenges of program abstraction These arise particularly from the scattering of information across a program, and the need to link related elements, such as the declaration and usages of a variable. Processing of large languages with many different program features is also a challenge.

RQ3: Design patterns/idioms We identified three key patterns:

- *Multi-pass processing*: one input AST *t* is processed by two or more rulesets to produce different parts of the target text.
- *Explicit grammar descent*: a user-defined function *r* is defined to process different forms of AST with different tags/arities and formats, such as unary and binary expressions. *r* is defined as a single ruleset which is recursively applied down an input AST structure.
- *Recursive term list iteration*: used when composite terms $(tag\ t_1\ \dots\ t_n)$ can be processed by processing the head t_1 of the subterms list and recursively invoking the same ruleset on the composite term $(tag\ t_2\ \dots\ t_n)$ built by taking the tail of the original subterm list.

RQ4: Transformation/grammar integration This is achieved by using the output (parse trees) of a language parser as input to the transformation. The transformation and grammar are closely related, because each grammar rule *tag* will produce parse trees (*tag t*1 ... *tn*) which need to be processed by the transformation (i.e., by a ruleset for *tag*).

7. Related Work

Related TTC cases are (1) [5] and (2) [4]. These concern (1) the extraction of state machines from Java code, and (2) the refactoring of Java code. An earlier case at GraBaTs '09 also concerned reverse engineering of Java for program comprehension [16]. This concerned the production of control flow and program dependence graphs.

The present case differs from these previous cases by (i) focussing on the fine-grained semantic modelling of program variables and data types, and (ii) by addressing a legacy source language (VB6) instead of Java. It also concerns program translation rather than comprehension or refactoring.

Specialised transformation approaches and languages have been utilised for program abstraction and re-engineering tasks: the TGraph concept and GReQL/GReTL languages are used for software migration in [3], and Gra2MoL for extracting models from code in [6]. These approaches have in common the need to effectively search and extract information from large graph or tree-structured program representations, which is a key task also in the present case. The present case however extends the scope of the abstraction task by requiring that a detailed semantic (mathematical) model is produced by abstraction, rather than specific search results or a syntactic (structural) model.

Conclusions

We have presented a challenging transformation case which involves the use of grammar-based and transformation tools for the reverse engineering and re-engineering of legacy code. The aim is to demonstrate that model transformations can be effective for this task, which is of high significance for industry.

References

[1] Eclipse, *Eclipse OCL*, https://wiki.eclipse.org/OCL/OCLinEcore, 2023.

[2] M. Eysholdt, H. Behrens, *Xtext: implement your language faster than the quick and dirty way*, OOPSLA 2010, pp. 307–309.

[3] A. Fuhr, T. Horn, V. Riediger, A. Winter, *Model-driven software migration into service-oriented architectures*, Comput. Sci. Res. Dev., vol. 28, 2013, pp. 35–84.

[4] M. Geza Kulcsar, S. Peldszus, M. Lochau, *Case Study: object-oriented refactoring of Java programs using graph transformation*, TTC 2015.

[5] T. Horn, *Program Understanding: a reengineering case for the Transformation Tool Contest*, TTC 2011, EPTCS.

[6] J. Izquierdo, J. Molina, *Extracting models from source code in software modernisation*, SoSyM vol. 13, 2014, pp. 713–734.

[7] R. Khadka et al., *How do professionals perceive legacy systems and software modernization?*, ICSE 2014, ACM Press, 2014.

[8] K. Lano, Q. Xue, S. Kolahdouz-Rahimi, *Agile specification of code generators for model-driven engineering*, ICSEA 2020.

[9] K. Lano, *Program translation using model-driven engineering*, short paper, ICSE 2022.

[10] K. Lano, Q. Xue, *Lightweight software language processing using ANTLR and CGTL*, Modelsward 2023.

[11] K. Lano, H. Haughton, Z. Yaun, *Program abstraction and re-engineering: an Agile MDE approach*, SAM 2023.

[12] Microsoft Com, *Office VBA Reference*, https://learn.microsoft.com/en-us/office/vba/api/overview, Oct. 2022.

[13] OMG, *Object Constraint Language 2.4 Specification*, OMG document formal/2014-02-03, 2014.

[14] R. Perez-Castillo, I. Garcia-Rodriguez de Guzman, M. Piattini, *Implementing business process recovery patterns through QVT transformations*, ICMT 2010.

[15] T. Sen, R. Mall, *Extracting finite-state representation of Java programs*, SoSyM, vol. 15 (2), 2016, pp. 497–511.

[16] J-S. Sottet, F. Jouault, *Program Comprehension Case*, GraBaTs 2009.