Generation of Multipurpose Formal Models from Legacy Code

Stepan Potiyenko[1], Alexander Kolchin[1]

[1]*V.M. Glushkov Institute of cybernetics of National Academy of Sciences of Ukraine, Academician Glushkov ave., 40, Kyiv, 03187, Ukraine*

Abstract

In this paper a method for generation of formal models from legacy software systems code is proposed. The purpose of these models is to have a possibility of their application in different tasks such as automatic generation of executable tests, translation to modern programming languages, reverse engineering. The method pursues goals to decrease complexity of state space search and checking formulas satisfiability, and to help legacy systems understanding and re-implementing using modern technologies. We focused on formalization of COBOL memory model as the most common in legacy systems. Formal model is an attributed transition system with arbitrary control flow. We propose an algorithm for building enumerated types for any variables whose usage fulfills certain conditions, including translation procedure of numeric variables into enumerated ones. We consider a problem of translating non-comparable structures that overlap in memory (operator *redefines* in COBOL) and are copied or compared with each other. In opposite to the common approach of using union semantics (like union construction in C++), our method of structure fields decomposition has no drawbacks of unions and contributes to minimization of the bytewise approach. We have examined the developed method using examples of structures with both simple fields and arrays. Examples of implementation of the bytewise approach in Java and C++ languages are given for those variables that cannot be represented as enumerated or numeric attributes. We have successfully applied this approach to generate tests on medium-sized projects (up to 100 000 lines of code). The generated formal models were also used to debug the Cobol to Java translator and to extract business rules.

Keywords

Translation, formal model, legacy systems.

Introduction

The problem of legacy code support is actual for several reasons — the number of specialists in old programming languages is insufficient and constantly decreasing, support for hardware systems (mainframe, etc.) is ending, while there is a need in the industry to develop new functionality and integrate with modern technologies. Mainly, the problem is solved by migrating old systems to new programming languages. Here a need for complete or partial automation arises, because, for example, according to estimates in 2021, from 200 to 250 billion lines of code in the COBOL language are running in the world [1]. Any automation begins with parsing the code and building some model with defined goals. Existing works, as a rule, pursue one of the goals - translation of legacy code into modern programming languages (COBOL translation into Java is very common), generation of executable tests, reverse engineering (code classification and clustering, information extraction, etc.). Models generated with a specific purpose, in addition to obvious advantages, have their disadvantages.

Conventional translators between languages generate intermediate models that are not very different from code. Their main goal is to present the constructions of the source code in a form convenient for further transformations. But the task of generating tests using such intermediate models has the same complexity as with any code. At the same time, these models are often

[1]13th International Scientific and Practical Conference from Programming UkrPROG'2022, October 11-12, 2022, Kyiv, Ukraine
EMAIL: stepan.potiyenko@gmail.com (A. 1); kolchin_av@yahoo.com (A. 2)
ORCID: 0000-0001-9462-599X (A. 1); 0000-0001-7809-536X (A. 2)

unreadable and look even less understandable for human than the source code. For example, the authors of [2] proposed a method of accurate translation of data types from legacy systems to Java classes. They produce 2 Java interfaces and 4 classes with several methods each (50-100 lines of code) to emulate two overlapping memory areas in COBOL language (7 lines with REDEFINES statement) using the semantics of the union construct from C++ language. To minimize a number of generated Java classes an algorithm for optimizing COBOL data structures based on similarities is given in [3]. Manual work is often required [4-7] in order to produce formal models from which automated test generation becomes possible.

Reverse engineering tools implement various abstraction techniques and are not intended for simulation. For the sake of a compact and clear presentation, a lot of information necessary for translators and test generators is lost. For example, the Rigi system [8] represents dependencies between code classes and functions in the form of a graph and divides it into subgraphs according to various criteria, performs information extraction but omits the control flow and other details. In [9], a method of slicing COBOL programs is proposed to help in understanding and maintaining systems. The authors of [10] build a formal model for visualizing a legacy system at different levels, from hardware to user interface. And in [11], a hypertext generator is proposed for navigating the data types of COBOL programs (type explorer), where memory intersections (REDEFINES) are considered as union.

The goal of our work is to generate formal models with the following properties:

- less complexity in relation to direct modeling of the code, which is important in such tasks as state space search, checking formulas satisfiability, test generation according to various coverage criteria, behavior analysis and debugging;
- accurate mapping of model artifacts to terms of source code to obtain executable tests and the possibility of translation into other languages;
- greater human understanding than the source code.

This work is an evolution of the methods and systems described in [12-14]. We pay the main attention to Cobol language, as it is the most widespread in the domain of legacy code, but we also had examples of applying our work for older versions of Java and Visual Basic. In this paper, we focus on the formalization of the memory model and also recall the general principles of building the model from the source code.

Formal model

As a formal model we use attributed transition system [17], where transitions are represented by tuples of the form (t,α,σ,β), where t is the name of the transition, α is its precondition, σ is an input or output signal, and β is a postcondition. The precondition contains the first-order predicate logic formula, and the postcondition contains a set of assignments of new values (expressions) to model attributes. Signals can contain parameters in the form of constants or attributes. The semantics of transitions is analogous to Dijkstra's guarded commands [15]: if the precondition of some transition t is fulfilled in some state s, then the model can perform this transition and pass to a new state $s'=t(s)$, which differs from the previous one by the values of the assigned attributes in postcondition. Model attributes are typed and can be integer, boolean or of enumerated types, and can also be arrays of elements of these types. We prefer the enumerated types in formal models: first, in order to increase the efficiency of symbolic computation, second, it significantly simplifies debugging and analysis of system behavior [13, 16].

To specify the control flow, directed graphs of the form $CFG = (V, E)$ are used, where V is a set of vertices, E is a set of edges specified by pairs of vertices. The vertices of the graph are either transitions of the formal model or links to other graphs that implement the semantics of function calls in the source code. Such a representation is comparable with the source code and is convenient for saving, human reading, and further translation of the formal model.

For a number of tasks, it is necessary to unfold all link-vertices by substitution into one *CFGU* graph. Such tasks include the determination of data dependencies, the construction of def-use pairs, and the search for cycles. Although explicit loops can be taken from the structure of the code, for example, in the COBOL language, GO TO operator and fall through semantics between paragraphs

are often used, which can generate implicit loops. When constructing a *CFGU* graph, there is a size problem because all reachable function call stacks must be expanded. It also makes it impossible to fully support recursion, although it can be limited, but this also leads to an increase in size. One method of solving this problem is to move the function call stack from the control flow to the data flow. But in this work, we do not solve this problem, taking into account the practical applications in which unfold approach was sufficient.

Expressions translation

Pre- and postconditions are represented in a form of abstract syntax trees (AST). Tree nodes express operators or terminal symbols. Operators in order of priority descending (from high to low) are the following:

- "[]" – access to an array element;
- " " – function call (typically, parentheses in source code);
- "abs" – integer modulus;
- "." – delimiter in full qualified names of fields of structures;
- "-", "!" – unary minus and negation;
- "*", "/" – arithmetic operations;
- "+", "-" – arithmetic operations;
- "<", ">", "<=", ">=", "==", "!=" – comparison operations;
- "&&" – conjunction (logical AND);
- "||" – disjunction (logical OR);
- "," – delimiter of parameters in function call;
- ":=" – assignment;
- ";" – delimiter of statements (currently used for several assignments in one CFG node);

Binary operators have right-side associativity besides binary minus and division – they are left-sided.

AST is independent on source code language and may contain any uninterpreted operators. We can abstract them by changing to nondeterministic assignments and obtain upper approximation. The same is fair for library functions, whose bodies are absent in code under analysis.

One of the difficulties in language transformations is a data types emulation. Let's consider main data definitions in COBOL from the Figure 1.

```
level-number [data-name-1 | FILLER]
  [REDEFINES data-name-2]
  [{PICTURE | PIC} IS character-string]
  [[USAGE IS] {BINARY | COMPUTATIONAL | COMP | DISPLAY | INDEX |
    PACKED-DECIMAL}]
  [OCCURS integer-2 TIMES
  [VALUE IS literal-1].
66 data-name-1 RENAMES data-name-2 [{THROUGH | THRU} data-name-3].
88 condition-name-1 {VALUE IS | VALUES ARE}
   {literal-1 [{THROUGH | THRU} literal-2]} ... .
```

Figure 1: Subset of COBOL data definitions syntax

Level numbers from 1 to 50 define hierarchy. A variable can appear to be a structure, which contains all variables defined below with greater level up to the end of section or to a variable with the same or lower level. There are special levels:

- level 66 is used to define alternative name data-name-1 of memory area containing given variable data-name-2 or all variables from data-name-2 to data-name-3;
- level 88 does not set any variable but is used for comparison of a variable defined above 88th level with defined value or a set of values.

Types of variables can be binary integer or alphanumeric, the latter we name string. We don't consider floating point types here.

We transform the names of 88th level as follows:

- All occurrences of names of 88th level with the only defined value are changed to comparison of parent variable with this value or to corresponding assignment, depending on semantics in source code.
- If a name of 88th level has a set of values every its occurrence is transformed to disjunction of corresponding comparisons with parent variable.

Enumerated types

Formal model contains a set T of enumerated types. Every type $T_i \in T$ is an unordered set of constants $T_i = \{e_1, \ldots, e_n\}$ with defined operations "==" (equal) and "!=" (not equal). These operations require strong type correspondence meaning that both arguments should have the same type. Variables in these arguments are called connected. For example, predicate $a == b[i]$ connects a variable a and an array b, so that a must be of the same type as elements of the array b, but it's not fair for index i. To generate first order predicate calculus formulas in a formal model we need to detect variables from source code, which can be represented as attributes of enumerated types, and build a set of these types.

Corresponding algorithm in a form of pseudo code with self-explanatory operators is the following:

```
// 1. Collect connected variables and their values from predicates
// they are comparison operators and assignments
for each variable v {
  new set CVALS(v) = collect values for v
  new set CVARS(v) = collect connected variables for v
}
// 2. Collect type groups
// they are pairs <set of variables, set of values>
new set G = ∅
for each variable v {
  if exists pair <VARS, VALS> ∈ G: v ∈ VARS
  then { // add v to existing group
    VARS = VARS ∪ {v} ∪ CVARS(v)
    VALS = VALS ∪ CVALS(v)
  } else { // add new group to G
    G = G ∪ {<{v} ∪ CVARS(v), CVALS(v)>}
  }
}
// 3. Build enumerated types
// collect pairs <set of attributes, set of enumerated elements>
new set T = ∅
for each pair <VARS, VALS> ∈ G {
  Boolean E = true
  for each variable v ∈ VARS {
    if not (check enumerated types restrictions for v)
    then E = false
  }
  if E
  then {
    T = T ∪ {<create set of attributes from VARS, create set of elements
from VALS>}
    G = G \ {<VARS, VALS>}
  }
}
```

Figure 2: Algorithm for building enumerated types

After completion we have two sets: T containing enumerated types in a form of pairs <set of attributes, set of elements of the type>, and G with groups of variables which do not fulfill restrictions for enumerated types. Variables from G become integer attributes or are processed bytewisely as described below.

Let's specify the algorithm for COBOL language:

1. Whole source code is analyzed and all predicates are collected. Predicates analysis allows to detect variables connected by operators requiring strong types correspondence (assignment, comparison). Also all occurring values are collected for each variable. Predicates analysis is performed by the following rules:

`MOVE VAL TO VAR1`
`VAR1 = VAL`
`VAR1 NOT = VAL`
 	
VAR1 is not a structure
VAL is a constant
	Constant VAL is added to the set of values CVALS(VAR1)
	This rule is also applied to predicates generated by procedure of variables decomposition.

`MOVE TERM TO VAR1(X:Y)`
`VAR1(X:Y) = TERM`
`VAR1(X:Y) NOT = TERM`
 	
VAR1(X:Y) is reference modification
	If X or Y is not a constant then the variable VAR1 is marked for bytewise processing. If it is a structure then all its fields are also marked as bytewise. If TERM is a name of a variable then it is also marked as bytewise (with all the fields in a case of structure).
	If both X and Y are constants then the variable VAR1 is decomposed by the algorithm described below.

`MOVE VAR2(X:Y) TO VAR1`
`VAR1 = VAR2(X:Y)`
`VAR1 NOT = VAR2(X:Y)`
 	
VAR2(X:Y) is reference modification
	If X or Y is not a constant then both variables (with all the fields in a case of structures) are marked for bytewise processing.
	If both X and Y are constants then the variable VAR2 is decomposed by the algorithm described below.

`MOVE TERM TO STR1`
`STR1 = TERM`
`STR1 NOT = TERM`
 	
STR1 is a structure
	If TERM is reference modification like VAR2(X:Y) then previous rule is applied.
	If TERM is a constant then we break it, as character string, into parts with lengths corresponding to STR1 fields. The predicate is transformed to a sequence of predicates (sequence of assignments, conjunction of equalities or disjunction of inequalities).
	
	If TERM is a variable name (can be a structure) then it is decomposed by the algorithm described below.

`MOVE STR2 TO VAR1`
`VAR1 = STR2`
`VAR1 NOT = STR2`
 	
STR2 is a structure
	Previous rule is applied where arguments of the predicate are swapped, i.e. TERM is VAR1 and STR1 is STR2.

`VAR1 > VAR2` Also operations >=, <, <=, NOT >, NOT <	If any variable VAR1 or VAR2 is not numeric by definition (PICTURE 9(N) or S9(N)) then both variables are marked for bytewise processing. It is also fair to structures.

Figure 3: Rules for analysis of COBOL statements

2. Connected variables are merged into groups so that all variables in each group have the same type. So, the set VALS of all found values of these variables forms a set of elements of corresponding enumerated type. Also auxiliary element OTHER is added for an abstract representation of other values which don't occur in code.
3. Enumerated type is generated for each group where all variables fulfill the following restrictions:

- no >, >=, <, <= comparisons with other variables or expressions (comparisons with numeric constants are allowed);
- no arithmetic operations;
- no occurrences of a variable as an array index;
- no operations over strings (like concatenation or substring), no substring operations with variable indices;
- no operator REDEFINES in DATA DIVISION (*)
- no operations over whole structures if a variable is a field of structure (*).

4. For groups where restrictions marked with asterisk (*) are not fulfilled, a method for variables decomposition is applied (described below). It produces new variables and new groups and this algorithm is repeated for them.

Names of attributes in the formal model are built as full qualified name of corresponding variable (names of all structures above this variable are appropriately joined).

Translation of numeric constants

Enumerated types are created for groups where variables are compared with numeric constants by operations >, >=, <, <= by the following algorithm:

1. Build a set of intervals I. Initially, $I = \{(-\infty,+\infty)\}$. Every constant c which occurs in comparison operations >, >=, <, <=, ==, != with a variable from current group, splits the interval from I which contains c to three: $c \in (n,m) \wedge (n,m) \in I \rightarrow I = (I \setminus \{(n,m)\}) \cup \{(n,c),[c,c],(c,m)\}$.
2. Create enumerated type T from the set of intervals I, where one and only one element corresponds to each interval.
3. For each predicate with comparison >, >=, <, <=, ==, != of a variable from current group with a constant, calculate a subset of allowed intervals from I and substitute this predicate with disjunction of equalities of the variable with corresponding elements of the type T.

For example, let we have predicates $v < 0$ and $v >= 5$. Then:

$$I = \{(-\infty,0), [0,0], (0,5), [5,5], (5,+\infty)\},\ T = \{\text{LS\_0, EQ\_0, GT\_0\_LS\_5, EQ\_5, GT\_5}\},$$
$$v < 0 \rightarrow v == \text{LS\_0},\ v >= 5 \rightarrow v == \text{EQ\_5} \,||\, v == \text{GT\_5}.$$

Numeric variables

Binary variables of Cobol language (BINARY, COMPUTATIONAL or COMP) are translated to integer attributes. According to the Cobol description, a binary variable with a PICTURE description of four or fewer decimal digits (S9(1) to S9(4) and 9(1) to 9 (4)) occupies 2 bytes; five to nine decimal digits, 4 bytes; and 10 to 18 decimal digits, 8 bytes. We abstract of the variables size while creating integer attributes, however, it is important for the bytewise approach.

Numeric variables (not binary) with PICTURE description 9(N) or S9(N) are translated to integer attributes in the same way as binary but differ in bytewise representation. Each digit occupies one byte with a code of digit symbol.

If all integer variables from a particular type group fulfill restrictions of enumerated types then an enumerated type and attributes are created.

Decomposition of variables

In the case when different structures are compared with each other, one is assigned to another, or they overlap in memory using the REDEFINES operator, it is necessary to bring them to one common structure. To work with structures overlapping in memory, it is common to use union semantics [2, 11], but this has the disadvantage of not being able to use two fields of different structures at the same time. In C++, the field to which the last assignment was made is considered active, and the behavior when reading inactive fields is undefined. In Java, there is no direct analogue of union, and even if to

make an implementation with data copying between fields, difficulties arise when the field types are not comparable. In general, for non-comparable structures, the bytewise approach is used, where, in one way or another, each byte of the variable is processed separately. This leads both to an increase in the number of states during modeling, and to the impossibility of reading and understanding such artifacts by a human. To avoid the bytewise approach, we suggest making the minimum necessary decomposition of the fields of the structures. For simplicity, it is better to show the algorithm on examples.

Example 1. Let we have two non-comparable structures STR1 and STR2 with lengths 5 and 6 that overlap in memory (starting from the same address). We will present a version of translation into a Java class using the bytewise approach:

```
01 STR1.
  05 A PIC X(2).
  05 B PIC X(3).
01 STR2 REDEFINES STR1.
  05 C PIC X(1).
  05 D PIC X(2).
  05 E PIC X(3).
```

```
public class STR1 {
  char[] data = new char[6];
  String getA() {
    return String.valueOf(data,0,2);
  }
  String setA(String s) {
    for (int i = 0; i < 2; i++)
      if (i < s.length())
        data[i] = s.charAt(i);
      else
        data[i] = ' ';
  }
  ...
  String getE() {
    return String.valueOf(data,3,3);
  }
  String setE(String s) {
    for (int i = 0; i < 3; i++)
      if (i < s.length())
        data[3 + i] = s.charAt(i);
      else
        data[3 + i] = ' ';
  }
}
```

Figure 4: Example of the bytewise approach in Java

Fields of these structures overlap in memory as shown in the Figure 5.

A		B			
C	D		E		

Figure 5: Fields overlapping in memory

Let's divide all fields of one structure by boundaries of fields of another, and vice versa, as in the Figure 6.

A_FIELD1	A_FIELD2	B_FIELD1	B_FIELD2		
C	D_FIELD1	D_FIELD2	E_FIELD1		E_FIELD2

Figure 6: Fields decomposition

Now, we can map fields of structures one-to-one and apply the algorithm for building enumerated types.

Besides overlaps in memory (REDEFINES), decomposition is required for statements STR1 = STR2 and MOVE STR1 TO STR2. To do this, align the structures along the left edge and make a similar decomposition, as well as transform the statements accordingly, as in the Figure 7.

`STR1 = STR2` is transformed to: `A_FIELD1 = C AND A_FIELD2 = D_FIELD1 AND … AND E_FIELD2 = SPACE`
`MOVE STR1 TO STR2` is transformed to: C := A_FIELD1; D_FIELD1 := A_FIELD2; …; E_FIELD2 := SPACE;

Figure 7: Statements transformation

When comparing and assigning alphanumeric variables of different lengths in Cobol, the shorter one is prolonged by spaces, hence the SPACE in the E_FIELD2 field.

Example 2. Take the structure STR2 from the example 1 and the variable N defined as 01 N PIC 9(4). In the case of numeric variables, they must be right-aligned and leading zeroes are provided for shorter one. Then, for the statements STR2 = N and MOVE N TO STR2, decomposition is shown in the Figure 8.

0	0	N_RFIELD2	N_RFIELD1		
C	D_FIELD1	D_FIELD2	E		

Figure 8: Fields decomposition

Analogous decomposition is applied by substrings boundaries in the statement VAR(X:Y) (reference modification) with constant indices. Next consider arrays.

Example 3. Let the structure STR1 contains an array of structures ARR1 with length of 3 elements, see the Figure 9.

```
01 STR1.
  05 ARR1 OCCURS 3 TIMES.
    10 A PIC X(1).
    10 B PIC X(2).
```

```
01 STR2.
  05 C PIC X(1).
  05 D PIC X(3).
  05 E PIC X(1).
  05 F PIC X(2).
```

Figure 9: Fields decomposition

In this case of decomposition it's required to unfold the array by elements as in the Figure 10.

A_1	B_1		A_2	B_2_FIELD1	B_2_FIELD2	A_3	B_3	
C	D_FIELD1		D_FIELD2	E	F_FIELD1	F_FIELD2		

Figure 10: Fields decomposition

In some cases, we can avoid unfolding arrays. We have developed a procedure for determining such cases and show one of them in the following example.

Example 4. Take the structure STR1 from the example 3 and the variable 01 S PIC X(12). Variable S is 12 bytes long, while the entire array ARR1 is 9 bytes. Then the variable S can be divided into a corresponding array and a tail of length 3, and the structure STR1 will remain unchanged, see the Figure 11.

```
01 STR1.
  05 ARR1 OCCURS 3 TIMES.
    10 A PIC X(1).
    10 B PIC X(2).
```

```
01 S PIC X(12).
Is transformed to a structure:
01 S.
  05 S_FIELD1 OCCURS 3 TIMES.
    10 S_FIELD1_1 PIC X(1).
    10 S_FIELD1_2 PIC X(2).
  05 S_FIELD2 PIC X(3).
```

Figure 11: Fields decomposition

In general, decomposition of variables allows to group new fields of different structures and build enumerated types for them. In the worst case, variables will be split into fields of one byte each, which is equivalent to the bytewise approach.

The bytewise approach

The bytewise approach is used in cases where the constraints of enumerated types are not met and integer attributes cannot be generated, for example:

- VAR(X:Y) (reference modification) – access to a substring with variable indices;
- STRING – operation of concatenation of several values into one;
- Comparisons >, >=, <, <= of non-numeric variables or structures requires lexicographical processing.

For bytewise processing, variables are represented as arrays of integers in the formal model. Each element of the array corresponds to one byte of memory.

The bytewise approach generates complex formulas or even behaviors implemented by source code language operators. We consider them as atomic transitions, and implement them as external functions in the C++ language, the calls of which are located in the preconditions and postconditions of model transitions. Thus, during simulation, we avoid generating many unnecessary states, but completely preserve the semantics of the source code. The Figure 12 shows the implementation of one of the simple functions cobol_move_int, which is used for bytewise processing of the construction MOVE N TO VAR, where N is an integer number, VAR is not a binary variable (there is another function for binary).

```
void cobol_move_int(int attr, int number) {
  // attr – identifier of an attribute in the model, the attribute must be an array of integers
  // number – a number to be written as a string
  string str = to_string(abs(number)); // convert modulus of the number to a string
  int lens = str.length(); // length of the string
  int len = get_array_size(attr); // length of the array
  int symb = 0;
  for (int i = 0; i < len; ++i) { // for each byte of the array attr
    if (i < len - lens)
      symb = '0'; // fill the beginning by zeroes
    else if (i == len - lens && number < 0)
      // set a flag of negative number according to Cobol semantics
      symb = str[0] | 0b01000000;
    else
      symb = str[i + lens - len]; // take current symbol of the string
    set_array_value(attr, i, symb); // write the symbol in the array
  }
  return 0;
}
```

Figure 12: Implementation of cobol_move_int function

Such external functions are implemented to simulate all cases of accessing variables in bytewise representation – for writing strings, filling with one character (LOW-VALUE, SPACE, ZERO, etc.), writing and extracting numbers in alphanumeric and binary representations, comparing alphanumeric, numeric binary and non-binary variables in various combinations, value type checks.

Conclusions

The proposed method of building formal models was applied to several medium-sized projects (10,000 – 100,000 lines of code) for further use in test generation [13, 14] and behavior analysis [12, 16] and showed its viability. The specifics of projects in the Cobol language is such that the definition of static memory (DATA DIVISION) occupies most of the code, so the generated models contain significantly fewer transitions than lines of code. The methods of constructing enumerated types and decomposing variables made it possible to effectively search the state space for the purpose of generating executable tests. Automated testing of the production version of the Cobol to Java

translator on a banking project was performed and 5 defects were found in the translator and 2 in the source code. On larger projects, the method of variables decomposition played a special role, it reduced the size of memory that is processed bytewisely, on average, by 3 times. This not only increased the efficiency of test generation and allowed to reach more coverage, but also significantly simplified the models for human understanding and reduced the time to debug the tests and the systems under testing.

References

[1] Patrick Stanard, A history of COBOL, why it's so popular today, where to find COBOL talent and the benefits of migrating to v6.3. https://techchannel.com/Enterprise/03/2021/business-systems-cobol. (2021)

[2] M. Ceccato, T.R. Dean, P. Tonella, D. Marchignoli, Data Model Reverse Engineering in Migrating a Legacy System to Java, Reverse Engineering, 2008. WCRE '08. 15th Working Conference on , vol., no., pp.177–186. (2008)

[3] Yohei Ueda, Moriyoshi Ohara. Refactoring of COBOL data models based on similarities of data field name. (2014)

[4] European Telecommunications Standards Institute. TTCN-3: Core Language. ES 201 873-1 4.11.1. (2019)

[5] International Telecommunications Union. Message Sequence Charts Z.120. (2011)

[6] A. Letichevsky, J. Kapitonova, V. Kotlyarov, V. Volkov, A. Letichevsky Jr., and T. Weigert, Semantics of Message Sequence Charts. Proc. 12th International SDL Forum: Model Driven, LNCS, vol. 3530, pp.117-132. (2005)

[7] M. Wynne and A. Hellesoy, The Cucumber Book. The Pragmatic Bookshelf. (2012)

[8] Holger M. Kienle, Hausi A. Müller, Rigi – An environment for software reverse engineering, exploration, visualization, and redocumentation, Science of Computer Programming, Volume 75, Issue 4, pp. 247-263. (2010)

[9] Ákos Hajnal & István Forgács, A demand-driven approach to slicing legacy COBOL systems. Journal of Software Maintenance, 24, pp. 67-82. (2012)

[10] A. Sivagnana Ganesan, T. Chithralekha, M. Rajapandian, A Formal Model for Legacy System Understanding. I.J. Intelligent Systems and Applications, 10, pp. 27-41. (2018)

[11] Arie van Deursen, Leon Moone, Exploring Legacy Systems Using Types. Proceedings Seventh Working Conference on Reverse Engineering. IEEE, pp. 32-41. (2000)

[12] A. Guba, et al., A method for business logic extraction from legacy COBOL code of industrial systems. In: Proceedings of the 10th International Conference on Programming UkrPROG2016, CEUR-WS, vol. 1631, pp. 17–25 (2016)

[13] T. Weigert, et al., Generating test suites to validate legacy systems. In: Fonseca i Casas, P., Sancho, M.-R., Sherratt, E. (eds.) SAM 2019. LNCS, vol. 11753, pp. 3–23. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30690-8_1

[14] A. Kolchin, S. Potiyenko, T. Weigert, Challenges for automated, model-based test scenario generation. Comm. Comput. Inf. Sci. 1078, 182–194. (2019)

[15] Edsger W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18.8 (1975), pp. 453-457. (1975)

[16] A. Kolchin, Interactive method for cumulative analysis of software formal models behavior. Proc. of the 11th Int. Conf. on Programming UkrPROG'2018, CEUR-WS vol. 2139, pp. 115–123. (2018)

[17] A. Letichevsky, A. Godlevsky, O. Letychevskyy (jr.), S. Potiyenko, V. Peschanenko, Properties of VRS predicate transformer. Cybernetics and System Analysis, vol. 46, pp. 521–532. (2010)