УДК 004.414.23+004.416.6+004.4’414+004.4’418 https://doi.org/10.15407/pp2022.03-04.042

ГЕНЕРАЦІЯ БАГАТОЦІЛЬОВИХ ФОРМАЛЬНИХ МОДЕЛЕЙ ІЗ УСПАДКОВАНОГО КОДУ

Степан Потієнко, Олександр Колчин

У роботі запропоновано метод генерації формальних моделей із коду успадкованих систем. Призначення таких моделей полягає в можливості застосування в різних задачах, таких як автоматична генерація виконуваних тестів, трансляція в сучасні мови програмування, зворотня інженерія. Робота переслідує цілі зменшення складності пошуку в просторі станів та перевірки виконуваності формул по відношенню до прямого моделювання коду, а також допомоги в розумінні успадкованих систем та їх імплементації в сучасних технологіях. Основна увага приділена формалізації моделі пам’яті мови Cobol, адже це найбільш поширена мова в успадкованих системах. Формальна модель є атрибутною транзиційною системою з потоком керування. Запропоновано алгоритм побудови перелічуваних типів для будь-яких змінних, використання яких відповідає певним умовам, в тому числі наведена процедура трансляції числових змінних в перелічувані. Розглянута проблема трансляції неспівставних структур даних, що перетинаються в пам’яті (оператор redefines мови Cobol), копіюються чи порівнюються між собою. На відміну від поширеного підходу використання семантики об’єднань (подібного до конструкції union в C++), розроблено метод, що полягає в розбитті полів структур і не має недоліків об’єднань та сприяє мінімізації побайтного підходу. Створений метод розглянуто на прикладах структур як з простими полями так і з масивами. Наведені приклади реалізації побайтного підходу в мовах Java та C++ для тих змінних, що неможливо представити у вигляді атрибутів перелічуваних або числових типів. Робота була випробувана для генерації тестів на проєктах середнього розміру (до 100 000 рядків коду), що показало ефективність розробленого методу, а згенеровані формальні моделі також були використані для відлагодження транслятору Cobol в Java та видобутку бізнес правил.

Ключові слова: трансляція, формальна модель, успадковані системи.

In this paper a method for generation of formal models from legacy 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 in relation to the direct code modeling, and to help legacy systems understanding and implementing in modern technologies. We focused on formalization of Cobol memory model as it is the most common language in legacy systems. Formal model is an attribute transition system with control flow. We proposed an algorithm for building enumerated types for any variables whose usage fulfills certain conditions, including translation procedure of numeric variables to enumerated ones. We considered a problem of translating non-comparable structures which overlap in memory (operator redefines in Cobol), are copied or compared with each other. In opposite to common approach of using union semantics (like union construction in C++), we described a method of structure fields decomposition which has no drawbacks of unions and makes for minimization of bytewise approach. We considered the developed method on the examples of structures as with simple fields as with arrays. We also gave examples of realization of bytewise approach in Java and C++ languages for those variables that cannot be represented as enumerated or numeric attributes. We tried this work for tests generation for middle-sized projects (up to 100 000 lines of code) where it showed efficiency of developed method, also generated formal models were used for debugging of Cobol to Java translator and business rules extraction.

Keywords: translation, formal model, legacy systems.

Вступ

Проблема підтримки успадкованого коду (legacy code) є актуальною з декількох причин — кількість спеціалістів в старих мовах програмування недостатня та постійно зменшується, завершується підтримка апаратних систем (mainframe тощо). Існує необхідність розробки нової функціональності та інтеграції з сучасними технологіями. Основним методом вирішення проблеми є переведення старих систем на нові мови програмування. Тут існує необхідність повної або часткової автоматизації, бо, наприклад, за оцінками 2021 року в світі працюють від 200 до 250 мільярдів рядків коду в мові Cobol [1]. Будь-яка автоматизація починається з розбору (parsing) коду і побудови деякої моделі з визначеними цілями. Існуючі роботи, як правило, переслідують одну з цілей – трансляція успадкованого коду в сучасні мови програмування (дуже поширена трансляція Cobol в Java), генерація виконуваних тестів, зворотня інженерія (reverse engineering – класифікація та кластеризація коду, видобуток інформації тощо). Моделі, згенеровані з конкретною метою, окрім очевидних переваг мають свої недоліки.

Так транслятори між мовами генерують проміжні моделі, які не дуже відрізняються від коду. Їхня головна мета – представити конструкції вихідного коду в зручному для подальших перетворень вигляді. Але задача генерації тестів на таких моделях має ту саму складність, що і з будь-якого коду. Водночас такі моделі зовсім нечитабельні, вони виглядають навіть менш зрозумілими для людини, ніж вихідний код. Наприклад, автори [2] запропонували метод точної трансляції типів даних із успадкованих систем в Java класи, в тому числі розглянули емуляцію областей пам’яті мови Cobol, що перетинаються (оператор REDEFINES), за допомогою семантики конструкції union із мови C++. В [3] наведений алгоритм оптимізації структур даних Cobol за ознаками схожості для мінімізації згенерованих Java класів. Для генерації тестів часто використовують моделі, описані в окремих мовах моделювання, що потребує ручної роботи [4-7].

Інструменти зворотньої інженерії реалізують різні методи абстракції і не мають на меті моделювання. Заради компактного і зрозумілого представлення втрачається багато інформації, необхідної для

трансляторів та тест генераторів. Наприклад, система Rigi [8] представляє залежності між класами та функціями коду у вигляді графу та розбиває його на підграфи за різними критеріями, видобуває інформацію, але упускає потік керування та інші деталі. В [9] пропонується метод слайсингу (slicing) Cobol програм для допомоги в розумінні та підтримці систем. Автори [10] будують формальну модель для візуалізації успадкованої системи на різних рівнях, від апаратного до інтерфейсу користувача. А в [11] запропоновано генератор гіпертексту для навігації за типами даних Cobol програм (type explorer), де перетини пам'яті (REDEFINES) розглядаються як union.

Мета нашої роботи – генерація формальних моделей, що мають такі властивості:

- менша складність щодо прямого моделювання коду, що проявляється в таких задачах, як пошук у просторі станів, перевірка виконання формул, генерація тестів за різними критеріями покриття, аналіз поведінки та налагодження;
- точне відображення артефактів моделі в терміни вихідного коду для отримання виконуваних тестів та можливості трансляції в інші мови;
- більше розуміння для людини, ніж у вихідного коду.

Дана робота є розвитком методів та систем, описаних в [12-14]. Головну увагу ми приділяємо мові Cobol, як найбільш поширеній у домені успадкованого коду.Але також мали приклади застосування своєї роботи на старих версіях Java та Visual Basic. В даній публікації зосередимось на формалізації моделі пам'яті, а також згадаємо загальні принципи побудови моделі з вихідного коду.

Формальна модель

Формальна модель містить потік даних та потік керування.

Для визначення потоку даних використовується атрибутна транзиційна система, в якій переходи представлені кортежами виду (t,α,σ,β), де t – ім'я переходу, α – його передумова, σ – вхідний або вихідний сигнал, β – постумова. Передумова містить формулу логіки предикатів першого порядку, а постумова – множину присвоєнь нових значень (виразів) атрибутам моделі. Сигнали можуть містити параметри у вигляді констант або атрибутів. Семантика переходів аналогічна охороняємим командам Дейкстри [15]: якщо передумова певного переходу t виконується в деякому стані s, то модель може його здійснити і перейти в новий стан $s'=t(s)$, який відрізняється від попереднього значеннями атрибутів, присвоєних у постумові. Атрибути моделі типізовані і можуть бути цілочисельними, бульовими або мати перелічувані типи, а також можуть бути масивами елементів цих типів. Ми віддаємо перевагу перелічуваним типам у формальних моделях: по-перше, з метою підвищення ефективності символьних обчислень, по-друге – це суттєво спрощує налагождення та аналіз поведінки системи [13, 16].

Для визначення потоку керування використовуються орієнтовані графи виду $CFG = (V, E)$, де V – множина вершин, E – множина ребер, які задаються парами вершин. Вершинами графа є або переходи формальної моделі або посилання на інші графи, які реалізують семантику викликів функцій у вихідному коді. Таке представлення співмірне з вихідним кодом і зручне для збереження, читання та подальшої трансляції формальної моделі.

Для ряду задач необхідно розгорнути всі вершини-посилання методом підставлення в один граф *CFGU*. До таких задач належать визначення залежностей даних (data dependencies), побудова пар визначення-використання атрибутів (def-use pairs) та пошук циклів. Нехай явні цикли можна взяти із структури коду, але, наприклад, в мові Cobol нерідко використовують оператор go to та семантику провалювання (fall through) між параграфами, що може породжувати неявні цикли. При побудові графу *CFGU* виникає проблема розміру, бо є потреба розгорнути всі досяжні стеки викликів функцій. Також унеможливлюється повна підтримка рекурсії, хоча її можна обмежити, але це теж призводить до зростання розміру. Одним із методів вирішення цієї проблеми є перенесення стеку викликів функцій із потоку керування до потоку даних. Але в даній роботі цю проблему ми не вирішуємо, зважаючи на практичні застосування, в яких було достатньо підходу розгортки.

Трансляція виразів

Перед- і постумови представляються у вигляді абстрактних синтаксичних дерев (АСД). Вершини дерева відображають оператори або термінальні символи. Наведемо список операторів у порядку зменшення пріоритету:

«[]» – доступ до елементу масиву;
« « – виклик функціїї (зазвичай, це круглі дужки у вихідному коді);
«abs» – модуль цілого числа;
«.» – роздільник в повних кваліфікованих іменах полів структур;
«-», «!» – унарний мінус та заперечення;
«*», «/» – арифметичні операції;
«+», «-» – арифметичні операції;
«<», «>», «<=», «>=», «==», «!=» – операції порівняння;
«&&» – кон'юнкція;
«||» – диз'юнкція;

«,» – роздільник у списку параметрів виклику функції;

«:=» – присвоювання;

«;» – роздільник у списку операторів, використовується у випадку декількох присвоювань в одній постумові;

Бінарні оператори мають правосторонню асоціативність окрім бінарного мінуса і ділення, вони лівосторонні.

АСД не залежить від мови вихідного коду і може містити неінтерпретовані оператори. Від них можна абстрагуватись введенням недетермінованих присвоювань і таким чином отримати верхню апроксимацію. Так само можна абстрагуватися від бібліотечних функцій, які викликаються в аналізованій програмі, але їхній код відсутній.

Труднощі викликає трансформація типів даних конкретної мови програмування в типи атрибутів формальної моделі. Розглянемо синтаксис визначення даних в мові Cobol:

```
level-number [data-name-1 | FILLER]
  [REDEFINES data-name-2]
  [IS EXTERNAL]
  [IS GLOBAL]
  [{PICTURE | PIC} IS character-string]
  [[USAGE IS] {BINARY | COMPUTATIONAL | COMP | DISPLAY | INDEX |
    PACKED-DECIMAL}]
  [[SIGN IS] {LEADING | TRAILING} [SEPARATE CHARACTER]]
  [OCCURS integer-2 TIMES
    [{ASCENDING | DESCENDING} KEY IS {data-name-3} ... ] ...
    [INDEXED BY {index-name-1} ...] |
   OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name-4
    [{ASCENDING | DESCENDING} KEY IS {data-name-3} ... ] ...
    [INDEXED BY {index-name-1} ...]]
  [{SYNCHRONIZED | SYNC} [LEFT | RIGHT]]
  [{JUSTIFIED | JUST} RIGHT]
  [BLANK WHEN ZERO]
  [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]} ... .
```

Рівні (level-number) від 1 до 50 визначають ієрархію. Змінна вважається структурою, якщо нижче за текстом визначені змінні з більшим рівнем. Вона містить усі такі змінні до кінця секції або до змінної такого ж, або нижчого рівня. Існують спеціальні рівні:

- рівень 66 використовується для завдання альтернативного імені data-name-1 області пам'яті, що містить змінну data-name-2 або всі змінні з data-name-2 до data-name-3;
- рівень 88 не визначає нової змінної, а використовується для порівняння змінної, визначеної над 88-м рівнем, із заданим значенням або множиною значень.

Типи змінних можуть бути цілочисельними в бінарному представленні або символьно-числовими (alphanumeric, в подальшому будемо називати їх рядками). Числа з плаваючою комою в даній роботі не розглядаються.

Імена 88-го рівня трансформуються наступним чином:

- усі входження імен 88-го рівня з одним заданим значенням у виразах замінюються на порівняння відповідної змінної із цим значенням, або на відповідне присвоювання, залежно від семантики оператору у вихідному коді;
- якщо змінна 88-го рівня задає множину значень, то її входження трансформуються у диз'юнкцію відповідних порівнянь.

Перелічувані типи

Формальна модель містить множину перелічуваних типів T. Кожен тип $T_i \in T$ є невпорядкованою множиною констант $T_i = \{e_1, \ldots, e_n\}$ із визначеними операціями «==» (рівність) and «!=» (нерівність). Ці операції потребують відповідності типів, тобто обидва аргументи повинні мати один тип, а змінні в цих аргументах називаються пов'язаними. Наприклад, предикат $a == b[i]$ пов'язує змінну a та масив b, тобто a по-

винна мати той самий тип, що й елементи масиву *b*, але індексу *i* це не стосується. Щоб згенерувати формули числення предикатів першого порядку в формальній моделі, необхідно виділити змінні з вихідного коду, які можна представити як атрибути перелічуваних типів, і побудувати множину цих типів.

Наведемо відповідний алгоритм у вигляді псевдо коду:

```
// 1. Зберемо пов'язані змінні та їх значення з предикатів.
// Це операції порівняння та присвоювання.
для кожної змінної v {
  створити множину CVALS(v) = зібрати значення для v
  створити множину CVARS(v) = зібрати змінні, пов'язані з v
}
// 2. Побудуємо групи змінних.
// Це пари <множина змінних, множина значень>
створити множину пар G = ∅
для кожної змінної v {
  якщо існує пара <VARS, VALS> ∈ G: v ∈ VARS
  тоді { // додати v до існуючої групи
    VARS = VARS  {v} ∪ CVARS(v)
    VALS = VALS ∪ CVALS(v)
  } інакше { // додати нову групу до G
    G = G ∪ {<{v} ∪ CVARS(v), CVALS(v)>}
  }
}
// 3. Побудуємо перелічувані типи.
// Зберемо пари <множина атрибутів, множина елементів перелічуваного типу>
створити множину пар T = ∅
для кожної пари <VARS, VALS> ∈ G {
  Boolean E = true
  для кожної змінної v ∈ VARS {
    якщо not (перевірити обмеження перелічуваних типів для v)
    тоді E = false
  }
  якщо E
  тоді {
    T = T ∪ {<створити множину атрибутів із VARS, створити множину елементів
із VALS>}
    G = G \ {<VARS, VALS>}
  }
}
```

В результаті роботи алгоритму маємо дві множини: T, яка містить перелічувані типи у вигляді пар <множина атрибутів, множина елементів типу>, і *G*, яка містить групи змінних, використання яких не відповідає обмеженням перелічуваних типів. Змінні з *G* транслюються в цілочисельні атрибути, або оброблюються побайтно, що описано нижче.

Наведемо специфікацію дій алгоритму для мови Cobol:

Предикати збираються по всьому вихідному коду. Аналіз предикатів дозволяє визначити змінні, пов'язані між собою операторами, що вимагають відповідності типів (присвоювання, порівняння). Також для кожної змінної збираються усі значення, що зустрічаються в коді. Предикати аналізуються за наступними правилами:

MOVE VAL TO VAR1 VAR1 = VAL VAR1 NOT = VAL – VAR1 – не структура VAL – константа	Константа VAL додається до множини значень CVALS(VAR1). Також це правило застосовується до предикатів, згенерованих в результаті розбиття змінних.
MOVE TERM TO VAR1(X:Y) VAR1(X:Y) = TERM VAR1(X:Y) NOT = TERM – VAR1(X:Y) – reference modification	Якщо X або Y не є константами, то змінна VAR1 позначається для побайтної обробки. Якщо це структура, то всі її поля теж позначаються як побайтні. Якщо TERM є іменем змінної, то і вона позначається як побайтна (з усіма полями у випадку структури). Якщо X та Y константи, то для змінної VAR1 застосовується алгоритм розбиття, наведений нижче.

`MOVE VAR2(X:Y) TO VAR1`
`VAR1 = VAR2(X:Y)`
`VAR1 NOT = VAR2(X:Y)`
–
`VAR2(X:Y) – reference modification`
	Якщо X або Y не є константами, то обидві змінні (з усіма полями у випадку структур) позначаються для побайтної обробки.
	Якщо X та Y константи, то для змінної VAR2 застосовується алгоритм розбиття, наведений нижче.

`MOVE TERM TO STR1`
`STR1 = TERM`
`STR1 NOT = TERM`
–
`STR1 – структура`
	Якщо TERM має вигляд VAR2(X:Y) (reference modification), то застосовується попереднє правило.
	Якщо TERM це константа, то вона, як рядок символів, розбивається на частини відповідно до довжин полів структури STR1, а предикат трансформується в послідовність предикатів (послідовність присвоєнь, кон'юнкція рівностей або диз'юнкція нерівностей).
	
	Якщо TERM є іменем змінної (може бути структурою), то застосовується алгоритм розбиття, наведений нижче.

`MOVE STR2 TO VAR1`
`VAR1 = STR2`
`VAR1 NOT = STR2`
–
`STR2 – структура`
	Застосовується попереднє правило, в якому аргументи предикатів помінялись місцями, тобто TERM – це VAR1, а STR1 – це STR2.

`VAR1 > VAR2`
`А також операції`
`>=, <, <=, NOT >, NOT <`
	Якщо будь-яка із змінних VAR1 або VAR2 не є числовою за описом (PICTURE 9(N) або S9(N)), то обидві змінні позначаються як побайтні. Це справедливо і для структур.

Із пов'язаних змінних формуються групи таким чином, що всі змінні в групі повинні мати один тип. Отже, множина VALS всіх знайдених значень цих змінних формує елементи відповідного перелічуваного типу. Також додається допоміжний елемент OTHER для абстрактного представлення інших значень, що не зустрічаються в коді.

Для кожної групи пов'язаних змінних створюється перелічуваний тип, якщо кожна змінна з групи задовольняє наступним обмеженням:

- немає порівнянь >, >=, <, <= з іншими змінними або виразами (порівняння з числовими константами допускаються);
- немає арифметичних операцій;
- змінна не зустрічається у вигляді індексу масиву;
- немає рядкових операцій (як конкатенація рядків), а також виділення підрядку із змінними індексами;
- немає оператору REDEFINES у визначенні даних (*);
- немає операцій над цілими структурами, якщо змінна є полем структури (*).

Для груп, в яких не виконуються обмеження, помічені зірочкою (*) в пункті 3, застосовується метод розбиття змінних (наведений нижче). Він продукує нові змінні і нові групи, для яких повторно виконується поточний алгоритм.

Імена атрибутів формальної моделі будуються як повне кваліфіковане ім'я відповідної змінної (додаються імена усіх вище стоячих структур).

Трансляція числових констант

Для груп, в яких змінні порівнюються з числовими константами операціями >, >=, <, <=, перелічувані типи створюються наступним чином:

Будуємо множину інтервалів I. На початку $I = \{(-\infty,+\infty)\}$. Кожна константа c, що зустрічається в операціях порівняння >, >=, <, <=, ==, != із змінною поточної групи, розбиває інтервал із множини I, якому вона належить, на три: $c \ \ (n,m) \wedge (n,m) \in I \rightarrow I = (I \setminus \{(n,m)\}) \cup \{(n,c),[c,c],(c,m)\}$.

Із множини інтервалів I формуємо перелічуваний тип T, де кожному інтервалу відповідає один і тільки один елемент.

Для кожного предикату, який містить порівняння >, >=, <, <=, ==, != змінної із поточної групи з константою, обчислюємо підмножину допустимих інтервалів з I, і замінюємо цей предикат на диз'юнкцію рівностей змінної з відповідними елементами типу T.

Наприклад, маємо предикати $v < 0$ та $v >= 5$. Тоді:

$I = \{(-\infty,0), [0,0], (0,5), [5,5], (5,+\infty)\}$, T = {LS_0, EQ_0, GT_0_LS_5, EQ_5, GT_5},

$v < 0 \rightarrow v$ == LS_0, $v >= 5 \rightarrow v$ == EQ_5 || v == GT_5.

Числові змінні

Бінарні змінні мови Cobol (BINARY, COMPUTATIONAL або COMP) транслюються в цілочисельні атрибути. За специфікацією мови, бінарна змінна, яка має 4 або менше десяткових цифри за описом PICTURE (від S9(1) до S9(4) та від 9(1) до 9(4)), займає 2 байти; від 5 до 9 цифр – 4 байти; від 10 до 18 цифр – 8 байт. Ми абстрагуємось від розміру змінних для створення цілочисельних атрибутів, але це має значення в побайтному підході.

Числові змінні (не бінарні) з описом PICTURE 9(N) або S9(N) транслюються в цілочисельні атрибути так само, як і бінарні, але їх обробка відрізняється в побайтному представленні, кожна цифра займає один байт, який містить її код як символ.

Якщо всі числові змінні з однієї групи, включаючи бінарні, задовольняють обмеженням перелічуваних типів, тоді створюється відповідний перелічуваний тип і атрибути.

Розбиття змінних

У випадку, коли різні структури порівнюються між собою, одна присвоюється іншій або вони перетинаються в пам'яті за допомогою оператору REDEFINES, необхідно привести їх до однієї загальної структури. Для роботи зі структурами, що перетинаються в пам'яті, поширено використання семантики об'єднання (union) [2, 11], але це унеможливлює використання двох полів різних структур одночасно. В C++ активним вважається те поле, якому було здійснене останнє присвоєння, а поведінка при читанні неактивних полів невизначена. В Java немає прямого аналогу union, і навіть, якщо зробити реалізацію з копіюванням даних між полями, виникають складності, коли типи полів неспівставні. В загальному випадку для неспівставних структур використовується побайтний підхід, де, в тому чи іншому вигляді, кожен байт змінної обробляється окремо. Це призводить як до росту кількості станів при моделюванні, так і до неможливості читання та розуміння таких артефактів людиною. Щоб уникнути побайтного підходу ми пропонуємо зробити мінімально необхідне розбиття полів структур. Для кращого розуміння покажемо алгоритм на прикладах.

Приклад 1. Нехай маємо дві неспівставні структури STR1 та STR2 з довжинами 5 та 6, які накладаються в пам'яті (починаються з однієї адреси). Наведемо варіант трансляції в Java клас із використанням побайтного підходу:

```
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] = ' ';
  }
}
```

Поля цих структур в пам'яті накладаються таким чином:

A		B			
C	D		E		

Розіб'ємо всі поля однієї структури по межах полів іншої, та навпаки:

A_FIELD1	A_FIELD2	B_FIELD1	B_FIELD2		
C	D_FIELD1	D_FIELD2	E_FIELD1		E_FIELD2

Тепер поля структур можна співставити одне до одного і застосувати до них алгоритм побудови перелічуваних типів.

Окрім перетинів у пам'яті (REDEFINES), розбиття буде потрібним для операторів STR1 = STR2 або MOVE STR1 TO STR2. Для цього умовно вирівняємо структури по лівому краю і зробимо аналогічне розбиття, а також відповідно перетворимо оператори:

STR1 = STR2 → A_FIELD1 = C AND A_FIELD2 = D_FIELD1 AND … AND E_FIELD2 = SPACE

MOVE STR1 TO STR2 → C := A_FIELD1; D_FIELD1 := A_FIELD2; …; E_FIELD2 := SPACE;

При порівнянні та присвоюванні буквено-цифрових (alphanumeric) змінних різної довжини в Cobol коротша подовжується пробілами, звідси SPACE в полі E_FIELD2.

Приклад 2. Візьмемо структуру STR2 з прикладу 1 і змінну N, визначену як 01 N PIC 9(4). У випадку числових змінних, їх треба вирівнювати по правому краю і для коротшої додаються нулі попереду. Тоді для операторів STR2 = N та MOVE N TO STR2 розбиття буде виглядати так:

0	0	N_RFIELD2	N_RFIELD1		
C	D_FIELD1	D_FIELD2	E		

Аналогічно застосовується розбиття по межах підрядків в операторі VAR(X:Y) (reference modification) з константними індексами. Далі розглянемо масиви.

Приклад 3. Нехай структура STR1 містить масив структур ARR1 довжиною 3 елементи:

```
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).
```

У цьому випадку розбиття необхідно розгорнути масив поелементно:

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		

В деяких випадках можна уникнути розгортання масивів. Ми розробили процедуру визначення таких випадків і покажемо один з них у наступному прикладі.

Приклад 4. Візьмемо структуру STR1 з прикладу 3 і змінну 01 S PIC X(12). Змінна S має довжину 12 байт, тоді як весь масив ARR1 займає 9 байт. Тоді змінну S можна розбити на відповідний масив і хвіст довжиною 3, а структура STR1 залишиться незмінною:

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

```
01 S PIC X(12).
Трансформується в структуру:
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).
```

Загалом розбиття змінних дає змогу згрупувати нові поля різних структур і побудувати перелічувані типи для них. У найгіршому випадку, змінні будуть розбиті на поля довжиною один байт кожне, що еквівалентно побайтному підходу.

Побайтний підхід

Побайтний підхід використовується у випадках, коли не виконуються обмеження перелічуваних типів і неможливо згенерувати цілочисельні атрибути, наприклад:

- VAR(X:Y) (reference modification) – доступ до підрядку із змінними індексами.
- STRING – операція конкатенації декількох значень в одне.
- Порівняння >, >=, <, <= нечислових змінних або структур потребує лексикографічної обробки.

Для побайтної обробки змінні представляються у вигляді масивів цілих чисел в формальній моделі. Кожен елемент масиву відповідає одному байту пам'яті.

Побайтний підхід породжує складні формули або навіть поведінки, що реалізують оператори мови вихідного коду. Ми розглядаємо їх як атомарні переходи, а реалізуємо у вигляді зовнішніх функцій в мові C++, виклики яких розташовані в перед- і постумовах переходів моделі. Таким чином, під час моделювання ми уникаємо породження багатьох непотрібних станів, але повністю зберігаємо семантику вихідного коду. Наведемо реалізацію однієї з простих функцій cobol_move_int, яка застосовується для побайтної обробки конструкції MOVE N TO VAR, де N – ціле число, VAR – не бінарна змінна (для бінарних є інша функція):

```
void cobol_move_int(int attr, int number) {
  // attr – ідентифікатор атрибуту в моделі, атрибут має бути масивом цілих чисел
  // number – число для запису у вигляді рядку
  string str = to_string(abs(number)); // конвертувати модуль числа в рядок
  int lens = str.length(); // довжина рядку
  int len = get_array_size(attr); // довжина масиву
  int symb = 0;
  for (int i = 0; i < len; ++i) { // для кожного байту масиву attr
    if (i < len - lens)
      symb = '0'; // заповнити початок нулями
    else if (i == len - lens && number < 0)
      // встановити прапор від'ємного числа згідно семантики Cobol
      symb = str[0] | 0b01000000;
    else
      symb = str[i + lens - len]; // взяти поточний символ рядку
    set_array_value(attr, i, symb); // записати символ в масив
  }
  return 0;
}
```

Такі зовнішні функції реалізовані для моделювання всіх випадків доступу до змінних у побайтному представленні – для запису рядків, заповнення одним символом (LOW-VALUE, SPACE, ZERO тощо), запису та добування чисел у буквено-цифровому та бінарному представленнях, порівняння буквенно-цифрових, цифрових бінарних та небінарних змінних у різних комбінаціях, перевірки типів значень.

Висновки

Запропонований метод побудови формальних моделей був застосований на декількох проєктах середнього розміру (10 000 – 100 000 рядків коду) з метою подальшого використання у генерації тестів [13, 14] та аналізу поведінки [12, 16] і показав свою життєздатність. Специфіка проєктів мови Cobol така, що більшу частину коду займає визначення статичної пам'яті (DATA DIVISION), тому згенеровані моделі містять значно менше переходів, ніж код рядків. Методи побудови перелічуваних типів та розбиття змінних уможливили ефективність здійснення пошуку в просторі станів з метою генерації виконуваних тестів. Було виконане автоматичне тестування продуктової версії транслятора Cobol в Java на проєкті з банківської сфери і знайдено 5 дефектів у трансляторі та 2 у вихідному коді. На більших проєктах особливу роль відіграв метод розбиття змінних, який зменшив обсяг пам'яті, що обробляється побайтно, в середньому втричі.. Це не тільки підвищило ефективність генерації тестів і дало можливість отримати більше покриття, а й суттєво спростило моделі для розуміння користувачем та зменшило час налагодження тестів та систем, що тестуються.

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. Ceccato, M., Dean, T.R., Tonella, P., Marchignoli, D., 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. Letichevsky A., Kapitonova J., Kotlyarov V., Volkov V., Letichevsky A. Jr., and Weigert T. Semantics of Message Sequence Charts. Proc. 12th International SDL Forum: Model Driven, LNCS, vol. 3530, pp.117-132. (2005)
7. Wynne M. and Hellesoy, A. 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. Hajnal, Ákos & Forgács, István, 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. Guba, A., 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. Weigert, T., 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. Kolchin, A., Potiyenko, S.,Weigert, T.: 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. Kolchin, A. 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)

Одержано 29.07.2022

Про авторів:

Потієнко Степан Валерійович
кандидат фізико-математичних наук, старший науковий співробітник
h-індекс в Scopus 3
22 українські публікації, 10 іноземних.
e-mail: stepan.mail@gmail.com

Колчин Олександр Валентинович
кандидат фізико-математичних наук, старший науковий співробітник
h-індекс в Scopus 4, https://orcid.org/0000-0001-7809-536X
34 українські публікації, 16 іноземних.
e-mail: kolchin_av@yahoo.com

Про місце роботи авторів:

Інститут кібернетики імені В.М. Глушкова НАН України.
03187, Україна, Київ, проспект Академіка Глушкова, 40.
Тел. +380 44 526 2008, Факс +380 44 526 4178, e-mail: incyb@incyb.kiev.ua

Прізвища та ініціали авторів і назва доповіді англійською мовою:
Potiyenko S.V., Kolchin A.V.
Generation of multipurpose formal models from legacy code

Прізвища та ініціали авторів і назва доповіді українською мовою:
Потієнко С.В., Колчин О.В.
Генерація багатоцільових формальних моделей із успадкованого коду