УДК 004.416

МЕТОД ИЗВЛЕЧЕНИЯ ЛОГИКИ ПОВЕДЕНИЯ ИЗ ПРОМЫШЛЕННОГО ПРОГРАММНОГО КОДА НА ЯЗЫКЕ КОБОЛ

А.А. Губа, А.В. Колчин, С.В. Потиенко

Цель работы – разработка комплекса инструментальных средств для автоматизации анализа и упрощения понимания поведения кода программных систем. Предложены методы трансляции, абстракции, отладки и построения тестов для языка Кобол. Разработана экспериментальная система, реализующая предложенные методы.
Ключевые слова: трансляция, Кобол, моделирование, абстракция, генерация тестов, отладка.

Мета роботи – розробка комплексу інструментальних засобів для автоматизації аналізу та спрощення розуміння поведінки коду програмних систем. Запропоновано методи трансляції, абстракції, відлагодження та побудови тестів для мови Кобол. Розроблено експериментальну систему, яка реалізує запропоновані методи.
Ключові слова: трансляція, Кобол, моделювання, абстракція, генерація тестів, відлагодження.

The purpose of this work is to develop a software tool for analysis automation and simplifying of understanding of software systems behavior. Methods for translation, abstraction, debugging and test generation for COBOL are proposed. We developed a software system, which implements the methods.
Key words: translation, Cobol, modeling, abstraction, test generation, debugging.

Введение

Большинство из списка Fortune 500 компаний имеют унаследованные системы, написанные на языке Кобол, который был популярен в 1960–1980-х годах. IDC (International Data Corporation) подсчитали, что на сегодняшний день существует более 200 миллиардов строк Кобол кода. По оценкам Dr. Dobb's Journal около 90% финансовых транзакций в мире обрабатывается кодом на Коболе, и 75 % коммерческой обработки данных написано на Коболе. Общая стоимость используемого в настоящее время кода на Коболе оценивается в 2 триллиона долларов США. Современная индустрия испытывает потребность в его поддержке, модернизации, добавлении новой функциональности или перепроектировании и переписывании на другой язык программирования; The Garnter Group предполагают, что 85 % этого кода требует миграции на современные платформы. Часто такой код не имеет ни проектной документации, ни поддержки авторов. Отсутствие спецификаций решаемых задач и комментариев, неструктурность программ, вызванная как особенностями языка, так и многочисленными поправками к исходному коду, специфическая модель данных, множество разнородных компонентов и неявных связей между ними (типичное приложение на языке Кобол включает в себя несколько программ, описания таблиц баз данных, экранных форм, файлов) – все это приводит к неоправданно высоким затратам времени на простую адаптацию программиста к программному окружению, пониманию алгоритмов и внутренних связей приложения. По оценкам фирм, занимающихся ручным переводом старых программ, доля этих затрат составляет от 50 до 90%. Развитие методов автоматического извлечения логики непосредственно из исходного программного кода – актуальная задача для решения проблем, связанных с сопровождением кода [1, 2].

Часто в качестве результата анализа кода формулируются правила, представляющие собой формулу предикатов первого порядка. Однако такое извлечение логики не учитывает темпоральных аспектов, в результате для систем с нетривиальным поведением такие правила не всегда адекватны коду. Например, для поведения «выполнять запрос до тех пор, пока не будут получены подтверждения всех транзакций», может быть извлечено правило «если существует неподтвержденная транзакция, то выполнить запрос», но при этом будет утрачено зацикливание.

Для извлечения логики поведения мы используем ряд приемов абстракций и аппроксимаций. Так, в качестве эффективного инструмента для отделения несущественной части кода от интересующей широко используется построение срезов программ [2, 3]. Срез может помочь разработчику найти и выделить из сложной программы необходимую ему функциональность, поместить соответствующую часть исходной программы в отдельный модуль и переиспользовать в дальнейшем, например, для перевода на современную языковую платформу. Нинг и др. [3] описывают набор инструментов для выделения компонентов из больших Кобол-систем. Вдобавок к обратным и прямым статическим срезам применяют методы частичных вычислений [4]. Довольно большую часть кода составляют вычисления, от которых можно абстрагироваться, не утратив существенную для логики часть. Мы предлагаем метод выделения таких участков и замены их недетерминированными присваиваниями.

В разделе «формальная модель» приведены основные определения математического аппарата, раздел «трансляция» описывает особенности абстракций и перевода языка Кобол в формальную модель. Далее на

примере описаны методы статического и динамического анализа модели, исследования информационных связей и достижимости; в завершении приведено описание метода порождения тестов.

Формальная модель

Для применения формальных методов преобразований и проверки достижимости мы используем модель атрибутной транзиционной системы, в которой переходы представлены тройками вида (t,α,β), где t – имя перехода, α – его предусловие, а β – постусловие [5]. Предусловием служит формула предикатов первого порядка, в постусловии выполняются присваивания атрибутам модели. Семантика переходов аналогична охраняемым командам Дейкстры [6]: если в некотором состоянии s предусловие перехода t истинно, то модель может выполнить этот переход и перейти в новое состояние $s' = t(s)$, которое отличается от предыдущего значениями тех атрибутов, которым было выполнено присваивание новых значений в постусловии. Для задания потока управления используется UCM (Use-Case Maps). Это язык высокого уровня, который позволяет разрабатывать и изучать модели систем со сложной структурой и поведением. UCM связывает поведение системы с ее архитектурой в формальной и визуальной форме. Описание модели на UCM представляется множеством карт, которые состоят из конечного множества путей. Пути содержат вершины и определяют порядок их прохождения. Вершины могут быть конструктами разных типов, которые используются для задания начала и конца путей, альтернатив, последовательный и параллельных композиций, соединения путей, таймеров, прерываний и их обработки. UCM модель можно представить четверкой $< \mathrm{U, S, E, R} >$, где U – множество ее элементов; $\mathrm{S} \subseteq \mathrm{U}$ – множество стартовых точек; $\mathrm{E} \subseteq \mathrm{U}$ – множество конечных точек; $\mathrm{R} \subseteq \mathrm{U} \times \mathrm{U}$ – отношение переходов, которое определяет достижимость одной вершины из другой. Язык UCM был рекомендован для разработки моделей программных систем Международным Телекоммуникационным Союзом в 2008 году и имеет стандарт Z151 [7]. Формальная семантика языка описывается в работе [8].

Трансляция

Построение UCM основано на использовании знаний о зависимостях исходной программы (PDG), при этом выполняются: автоматическое преобразование процедур в соответствующие карты UCM диаграмм; преобразование «goto» операторов в пути между картами; трансляция структур данных, заданных в Procedure Division, Data Division, Copy-books; удаление данных, которые отсутствуют в условиях и сигналах; удаление заведомо недостижимого кода; разворачивание REDEFINE операторов; преобразование CICS операторов в сигналы; преобразование SQL выражений в сигналы.

Для анализа и моделирования программ строится граф потока управления (CFG) с сохранением структуры исходного кода. Он представляется древовидной структурой данных, которая для удобства записывается в формате JSON, который включает в себя объекты следующих типов: «start» – начальная точка потока управления; «return» – оператор возврата; «decision» – разветвление потока управления, имеет вложенный массив объектов, каждый из которых соответствует ветке (branch) потока управления; «branch» – условие ветки потока управления; «assignment» – выражение с присваиванием; «call» – вызов программы или процедуры; «readevent» – оператор чтения (из файла, терминала, т. д.); «writeevent» – оператор записи (в файл, терминал, т. д.); «goto» – оператор безусловного перехода к произвольной точке программы; «attribute» – определение переменной в программе. Также строится соответствие строкам исходного кода.

Выражения преобразуются в абстрактное синтаксическое дерево (AST), которое представляется как массив объектов. Первый объект всегда строка, определяющая оператор или терминальный символ. Остальные объекты могут быть как строками для задания терминалов, так и массивами объектов для вложенных AST.

Ниже представлены приоритеты операторов в убывающем порядке:

"[ ]" – обращение к элементу массива;

" " – вызов функции (скобки);

"-", "!" – унарный минус и отрицание;

"*", "/" – арифметические операции;

"+", "-" – арифметические операции;

"<", ">", "<=", ">=", "==", "!=" – операции сравнения;

"&&" – конъюнкция (логическое И);

"||" – дизъюнкция (логическое ИЛИ);

"," – запятая, разделяющая параметры в вызове функции;

":=" – присваивание.

Бинарные операторы имеют правостороннюю ассоциативность, кроме бинарного минуса и деления – они левосторонние.

AST не зависит от языка исходного кода и может содержать любые неспецифицированные операторы, как например, конкатенация строк. Так как в большинстве случаев такие операторы не влияют на логику программы, то можно от них абстрагироваться, заменив недетерминированными присваиваниями, тем самым получить верхнюю аппроксимацию. То же относится и к библиотечным функциям, тела которых отсутствуют в анализируемом коде.

Решение проблем, связанных с неожиданными побочными эффектами типов данных, – это одна из трудностей языковых преобразований. Для корректной трансляции необходим подход так называемой эмуляции типов данных. Рассмотрим синтаксис описания данных языка Кобол [9] (рис. 1).

```
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]} ... .
```

Рис.1. Синтаксис описания данных в языке Кобол

Номера уровней (level-number) от 01 до 50 задают иерархию. Переменная является структурой и содержит все переменные, объявленные ниже с более высоким уровнем до конца секции или до переменной с тем же или более низким уровнем. Есть специальные уровни:

– уровень 66 используется для задания альтернативного имени data-name-1 области памяти, содержащей заданную переменную data-name-2 или все переменные от data-name-2 до data-name-3;

– уровень 77 обозначает независимую переменную без возможности дальнейшего разбиения;

– уровень 88 не определяет переменных, а используется для сравнения переменной, определенной перед 88 уровнем, с заданным значением или множеством значений.

Тип любой переменной может быть числовым, буквенным или буквенно-цифровым. Последние два будем называть строковыми.

Для построения формул исчисления предикатов первого порядка необходимо преобразовать данные исходного языка в переменные перечислимых типов, целочисленные или булевские переменные. Разные уровни иерархии (поля структур) приводятся к простым переменным путем переименования. Строковые переменные преобразуются в переменные перечислимых типов по следующему алгоритму:

1. По всему исходному коду собираются пары переменных, связанных операторами, требующими строго соответствия типов (присваивание, сравнение), а также встречающиеся значения всех переменных.

2. Из собранных пар, по транзитивности строятся группы переменных, обязанных иметь один и тот же тип, а собранные значения этих переменных формируют элементы этого типа.

Такое преобразование сохраняет свойства достижимости исходного кода в абстрактной модели, если отсутствуют операции над строками.

Трансляция данных для 77-го уровня тривиальна. 88-й уровень преобразовывается двумя способами:

1. Если значения в разных именах 88-го уровня для одной переменной не пересекаются, то все вхождения этих значений в модели заменяются на соответствующие имена, которые и составляют перечислимый тип.

2. Если значения пересекаются (это типично, когда для некоторых имен 88-го уровня заданы множества значений), то верхняя переменная преобразуется в булевский функциональный символ с одним параметром перечислимого типа, построенного как в предыдущем пункте. Это позволяет иметь истинное значение переменной одновременно для нескольких имен 88-го уровня.

Дополнительные сложности возникают при несоблюдении типов, т. е. записи строковых значений в числовые переменные и наоборот, что вполне допускается в языке Кобол. Однако семантика операций над переменными, имеющими значения несоответствующего типа, не определена. Для обработки таких ситуаций мы генерируем сразу 3 переменных: целочисленная, перечислимого типа и переменная-флаг, сохраняющая информацию о том, какая из двух предыдущих модифицировалась последней. А при каждом чтении оригинальной переменной (т. е. одной из сгенерированных с учетом типа) также проверяется флаг, и в случае несоответствия выполняется обработка ошибки (сообщение пользователю или соответствующая трасса).

Некоторые системы трансляции языка Кобол выполняют преобразования области данных Кобол-программы в один массив памяти, а доступ к каждой переменной заменяют на ряд доступов к соответствующим элементам этого массива (побайтно). Такой подход может быть эффективным для одноразовой автоматической трансляции, но к сожалению, не пригоден для случая, когда требуется дальнейшая модификация результирующего кода и совсем неприемлем для извлечения логики.

Далее мы предлагаем рассмотреть методы преобразований и абстракций на сквозном примере, исходный текст кода которого показан на рис. 2. Граф зависимости для этой программы показан на рис. 3.

На рис. 4, а и 4, б изображен UCM, который автоматически построен из программы. Заметим, что строки 11 и 28 удалены ввиду того, что атрибут `Q` не используется в условиях и сигналах.

```
01. IDENTIFICATION DIVISION.
02. PROGRAM-ID. Example initial.
03. DATA DIVISION.
04. WORKING-STORAGE SECTION.
05. 01 BUTTON-RECORD.
06.    05 S PIC 9(16).
07.    05 KEYV PIC X(16).
08. 01 RESULT PIC X(16).
09. 01 X PIC 9(7) VALUE 'UNKNOWN'.
10. 01 Z PIC X(16).
11. 01 Q PIC 9(16).
12. 01 U PIC 9(16).
13. 01 FORM-RECORD.
14.    05 LOGIN PIC X(16).
15.    05 PASS PIC X(16).

16. PROCEDURE DIVISION.
17. EXEC CICS
18.     READ FILE ('F')
19.      INTO (BUTTON-RECORD)
20. END-EXEC.
21. IF EIBCALEN > ZERO
22.    MOVE S TO U
23. ELSE
24.    MOVE 'ABORT' TO RESULT
25.    EXIT
26. END-IF.
```

```
27. IF S > 0 THEN
28.    MOVE 1 TO Q
29.    IF KEYV = 'M' THEN
30.         MOVE 'MODERATOR' TO Z
31.    ELSE
32.         MOVE 'USER' TO Z
33.    END-IF
34.    IF U = 0 THEN
35.         MOVE Z TO X
36.    END-IF
37. ELSE
38.    PERFORM 0000-AUTH
39.    IF RESULT = 'OK' THEN
40.       MOVE 'ADMIN' TO X
41.    ELSE
42.       MOVE 'INTRUDER' TO X
43.    END-IF
44. END-IF.
45. EXEC CICS
46.    SEND TEXT FROM (X)
47. END-EXEC.
48. STOP RUN.

49. 0000-AUTH.
50. EXEC CICS READ
51.    FILE ('FORM')
52.    INTO (FORM-RECORD)
53. END-EXEC.
54. IF LOGIN NOT EQUAL 'admin' THEN
55.    MOVE 'FAILED' TO RESULT
56. ELSE
57.    IF PASS NOT EQUAL 'admin' THEN
58.       MOVE 'FAILED' TO RESULT
59.    END-IF
60. END-IF.
```

Рис. 2. Исходный код на языке Кобол

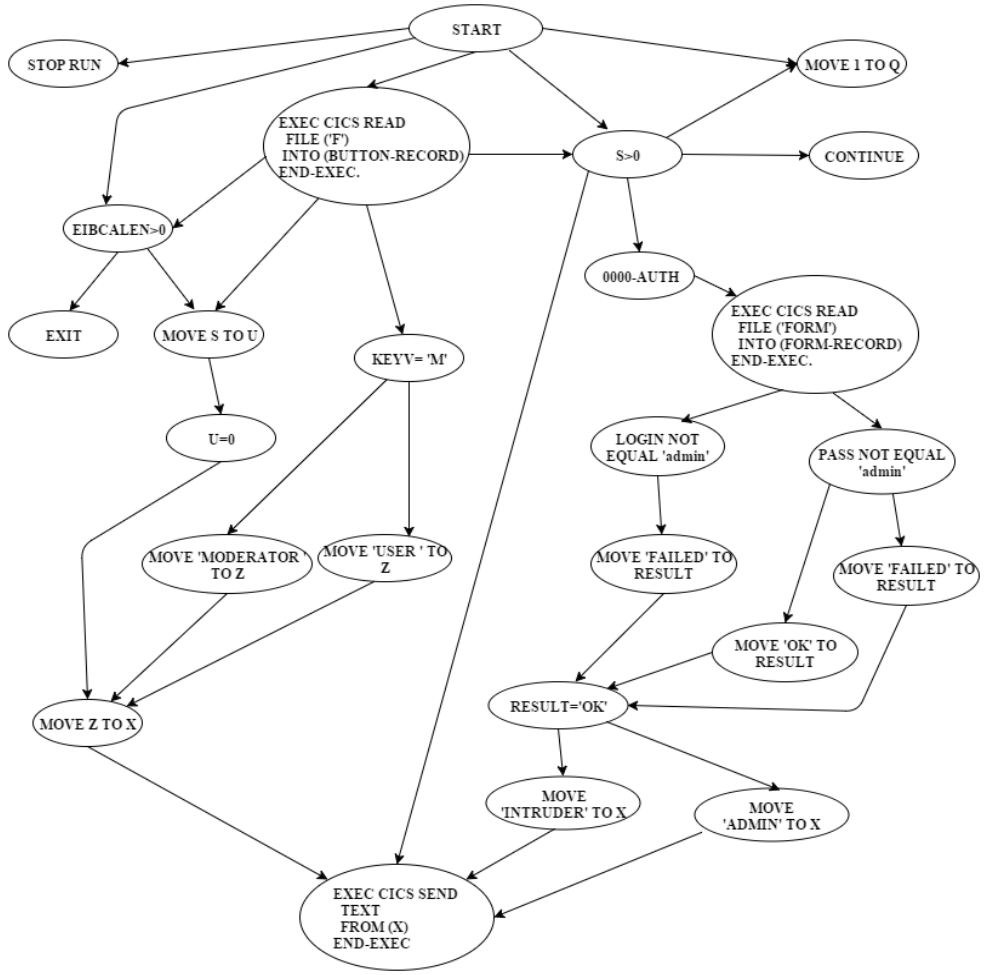
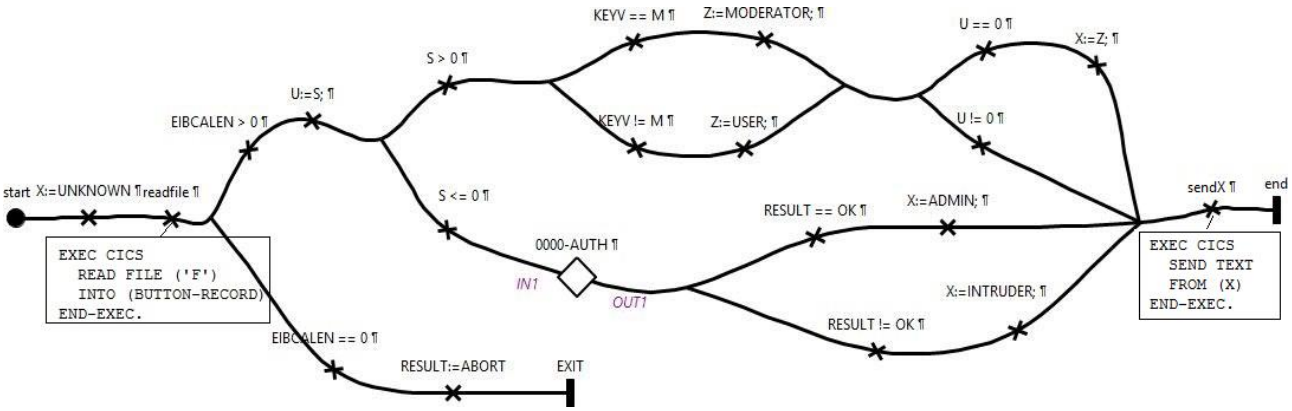
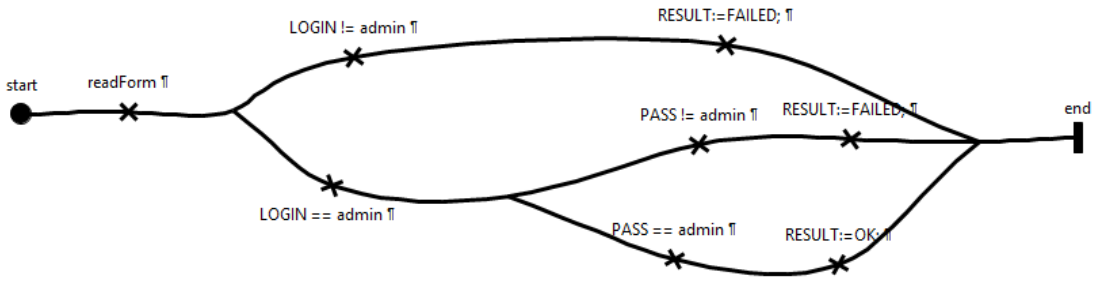


Рис. 3. Граф зависимостей в программе



а



б

Рис. 4. а – UCM для исходного кода, б – UCM для исходного кода процедуры 0000-AUTH

Статический срез

Исходная модель может быть очень большой. Извлечение логики может потребовать сильных огрублений. Но можно рассматривать поведение «по частям», например, организовав разбиение по принципу входных и выходных сигналов, исследуя их по отдельности. Так, задав множество интересующих сигналов, можно статически идентифицировать место их вхождения в коде и использовать технику слоев. Например, для выходных сигналов использовать обратные срезы, а для входных – прямые. Таким образом, можно восстановить логику зависимости выходных сигналов от входных. При этом размер кода будет существенно уменьшен. Так же можно прибегнуть к технике частичных вычислений. В результате построения обратного статического среза для строки 46 будет удалена строка 24.

Абстракция процесса вычислений

Часто код описывает довольно громоздкий анализ некоторых входных параметров, в результате которого вычисляется некое результирующее значение. Во многих случаях можно выделить процесс вычисления и исследовать его отдельно, а в общей схеме поведения, в целях уменьшения размера описания логики поведения, заменить его недетерминированным присваиванием, таким образом, абстрагироваться от загромождающих деталей. Интуитивно, достаточным условием корректности такой абстракции будет нарушение связности DEF-USE графа при удалении ребра, соединяющего процесс вычисления с результатом. На рис. 5 показан результат применения такого типа абстракции для строки 46.

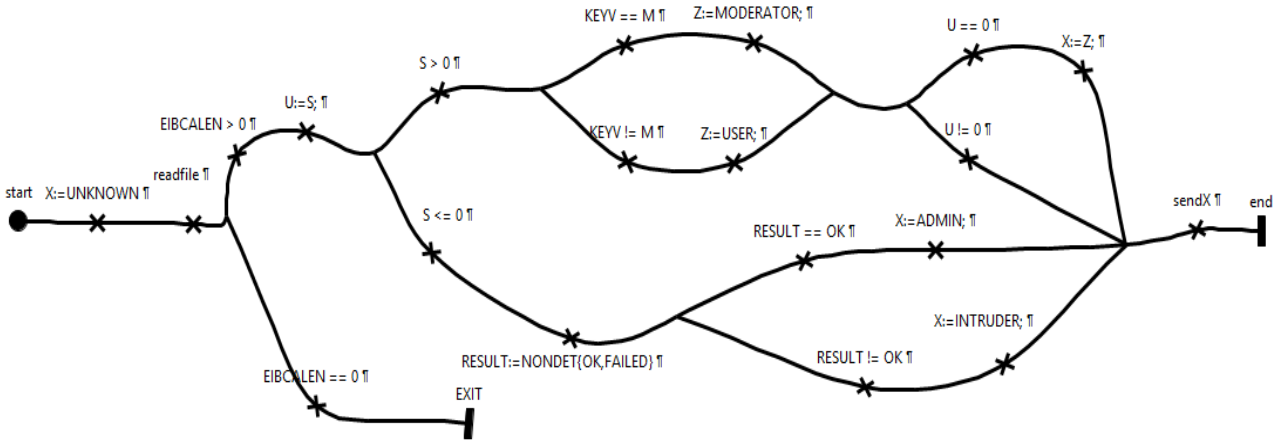


Рис. 5. Абстракция вычисления значения атрибута RESULT

На этом этапе удалены строки 49–60, строка 38 заменена строкой 38a, в которой используется оператор NONDET, обозначающий недетерминированный выбор.

Анализ достижимости

Как известно, статические методы преувеличивают фактические информационные связи [10], как следствие, некоторые несущественные для логики поведения участки кода могут остаться. Их можно удалить, прибегнув к использованию методов проверки достижимости. В нашем примере строка 35 недостижима, и после ее удаления и повторного выполнения процедуры обратного статического среза строки 29–33 будут удалены как несущественные (см. рис. 6). Таким образом, повтор процедуры статического среза после анализа достижимости может устранить зависимости, обусловленные недостижимыми переходами.

Недостижимым код может оказаться не только вследствие ошибки, но и после выполнения частичных вычислений, когда удаляются участки кода, не удовлетворяющие некоторому заданному условию. При этом некоторый код, подлежащий удалению, может остаться из-за того, что некоторые зависимости или условия не удалось разрешить статически.

Отметим полезную функциональность разработанной системы – оперативное выделение всех DEF-USE ветвей для заданного атрибута в заданной точке, причем с информацией об их достижимости. На базе этой функции реализован режим объяснения причин недостижимости. В этом режиме будут построены (и выделены на UCM специальной подсветкой) участки путей от последнего присваивания атрибуту, значение которого не удовлетворяет формуле предусловия недостижимого перехода, к самому переходу.

Следует иметь ввиду, что даже при условии конечности множества состояний модели, анализ достижимости может оказаться весьма трудоемким процессом ввиду так называемого эффекта комбинаторного взрыва [11, 12]. Поэтому приступать к этой процедуре лучше после применения статических абстракций, которые часто сокращают исходный размер на порядки.

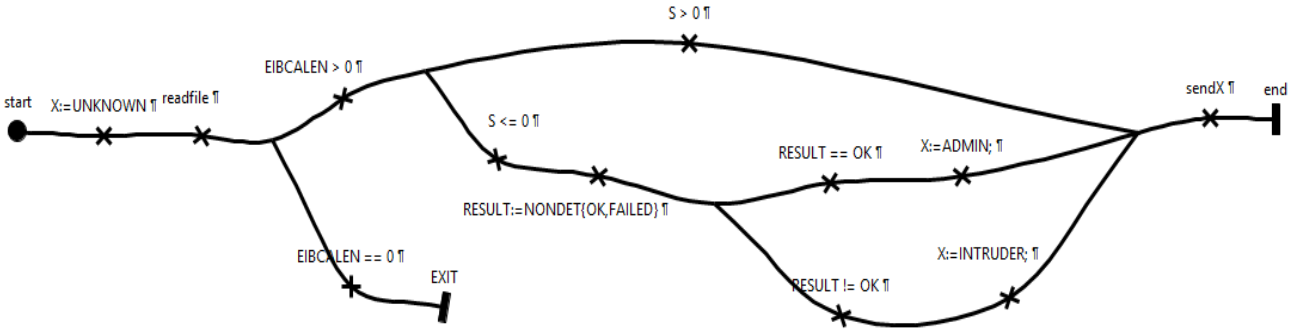


Рис. 6. Результат повторного статического среза

Результирующий код показан на рис. 7 и выглядит таким образом:

```
01. IDENTIFICATION DIVISION.
02. PROGRAM-ID. Example reduced.
03. DATA DIVISION.
04. WORKING-STORAGE SECTION.
05. 01 BUTTON-RECORD.
06.    05 S PIC 9(16).
08. 01 RESULT PIC X(16).
09. 01 X PIC 9(7) VALUE 'UNKNOWN'.
16. PROCEDURE DIVISION.
17. EXEC CICS
18.    READ FILE ('F')
19.     INTO (BUTTON-RECORD)
20. END-EXEC.
21. IF EIBCALEN > ZERO
22.     MOVE S TO U
23. ELSE
25.     EXIT
26. END-IF.
```

```
27. IF S > 0 THEN
27a.     CONTINUE
37. ELSE
38a.     MOVE NONDET{OK,FAILED} TO RESULT
39.      IF RESULT = 'OK' THEN
40.          MOVE 'ADMIN' TO X
41.      ELSE
42.          MOVE 'INTRUDER' TO X
43.      END-IF
44. END-IF.

45. EXEC CICS
46.    SEND TEXT FROM (X)
47. END-EXEC.
48. STOP RUN.
```

Рис. 7. Результирующий код после повторного статического среза

UCM сценарий, получившийся в результате описанных преобразований и абстракций описывает логику поведения модели, ведущего к интересующей точке (в нашем примере, к 46-й строке кода). Аналогично, можно исследовать влияние некоторой точки на последующее поведение, используя прямые срезы и абстракции.

Генерация тестов

При выполнении миграции на другую платформу возникает проблема проверки адекватности новой реализации. Одним из методов, широко применяемых на практике для этих целей, является тестирование. В условиях отсутствия спецификаций на исходную реализацию, актуальной становится задача порождения тестовых наборов. Некоторые подходы предлагают осуществлять запуск старой программы и фиксировать реакцию на различные входы вручную, однако такой подход не эффективен для больших систем. Мы предлагаем автоматическую генерацию тестовых сценариев на основе анализа кода. Так, построив формальную модель UCM, применяем существующие методы анализа достижимости, которые в процессе обхода поведения модели способны [11, 13] генерировать трассы (последовательности событий), которые представляют собой тестовые сценарии в виде Message Sequence Chart, после чего метод конкретизации [14] транслирует такие сценарии в исполнимые тесты. Проблемы комбинаторного взрыва и качества покрытия при генерации трасс заслуживают отдельного рассмотрения. Методы [10–13] эффективно строят классы эквивалентности для различных путей выполнения программ на основе статического [10] и динамического [11] анализа информационной зависимости, значительно сокращая перебор вариантов поведения, достигая при этом покрытия DEF-USE ветвей.

Примеры использования

Разработанный программный комплекс был успешно применен в ряде промышленных проектов. Далее представлена сводная таблица статистики.

Отметим, что операционное время, затраченное созданным программным комплексом на получение указанных результатов, исчисляется секундами. Такая производительность позволяет использовать инструментарий интерактивно в режиме реального времени, что очень востребовано при анализе поведения больших объемов программного кода.

Таблица. Статистика применения

Проект (домен)	Кол-во строк исходного кода (в тыс.)	Размер UCM (карт/конструктов)	Кол-во тестов
№ 1 (лизинг)	12.5	25/278	212
№ 2 (страхование)	17	34/330	364
№ 3 (финансы)	21	43/463	487

Выводы

Представленные методы эффективно устанавливают взаимосвязи в программном коде, выделяют релевантные участки, устраняют несущественные детали и загромождающие вычисления, оставляя семантически важную часть и предоставляя ее в наглядной визуальной форме. Результирующая UCM диаграмма (с соответствием строкам исходного кода) служит доступным объяснением искомой логики поведения.

Разработана экспериментальная вычислительная система, реализующая алгоритмы извлечения бизнес-логики программ на языке Кобол. Планируется усовершенствовать описанные методы для применения к программному коду на языках VBA, Java, C#.

1. *Cosentino V., Cabot J., and oth.* Extracting Business Rules from COBOL: A Model-Based Tool // Working Conference on Reverse Engineering, Koblenz, Germany. – 2013. – P. 483–484.
2. *Hajnal A., Forgacs I.* A demand-driven approach to slicing legacy COBOL systems // Journal of software: evolution and process. – Vol. 24. – P. 67–82.
3. *Ning J.Q., Engberts A., Kozaczynski W.* Automated support for legacy code understanding // Communs ACM. – 1994. – Vol. 38. – P. 50–57.
4. *Jones N., Gomard C., and Sestoft P.* Partial Evaluation and AutomaticProgram Generation. Prentice Hall International. – 1993. – 415 p.
5. *Летичевский А.А., Годлевский А.Б., Потиенко С.В.* Свойства предикатного трансформера системы VRS // Кибернетика и системный анализ. – 2010. – № 4. – С. 3–16.
6. *Dijkstra E.* Guarded commands, nondeterminacy and formal derivation of programs // Communications of the ACM. – 1975. – Vol. 18. – N 8. – P. 453–457.
7. *ITU-T* Recommendation Z.151 . User requirements notation (URN), 10/2012.
8. *Губа А.А., Шушпанов К.И.* Инсерционная семантика плоских многопотоковых моделей языка UCM // УСИМ. – 2012. – № 6. – С.15–22.
9. *Vale M.* The evolving algebra semantics of Cobol. Part 1: programs and control. Technical Report CSE-TR-162-93, EECS Dept., University of Michigan. – 1993. – 29 p.
10. *Колчин А.В., Летичевский А.А., Потиенко С.В.* Статический метод устранения избыточных информационных связей в предусловиях переходов формальных моделей транзиционных систем // Искусственный интеллект. –2015. – № 1–2. – С. 127–136.
11. *Колчин А.В.* Автоматический метод динамического построения абстракций состояний формальной модели // Кибернетика и системный анализ. – 2010. – № 4. – С. 70–90.
12. *Колчин А.В.* Метод редукции анализируемого пространства поведения при верификации формальных моделей распределенных программных систем // Искусственный интеллект. – 2013. – № 4. – С. 113–126.
13. *Колчин А.В., Котляров В.П., Дробинцев П.Д.* Метод генерации тестовых сценариев в среде инсерционного моделирования // Управляющие системы и машины. – 2012. – № 6. – С. 43–48, 63.
14. *Drobintsev P., Kolchin A., Kotlyarov V., Letichevsky A., Peschanenko V.* An approach to creating concretized test scenarios within test automation technology for industrial software projects // Automatic Control and Computer Sciences. –2013. – Vol. 47(7). – P. 433–442.

References

1. Cosentino V., Cabot J., and oth. Extracting Business Rules from COBOL: A Model-Based Tool // Working Conference on Reverse Engineering, Koblenz, Germany. – 2013. – P. 483–484.
2. Hajnal A., Forgacs I. A demand-driven approach to slicing legacy COBOL systems // Journal of software: evolution and process. – Vol. 24. – P. 67–82.
3. Ning J.Q., Engberts A., Kozaczynski W. Automated support for legacy code understanding // Communs ACM –1994. – Vol. 38. – P. 50–57.
4. Jones N., Gomard C., and Sestoft P. Partial Evaluation and AutomaticProgram Generation. Prentice Hall International – 1993. – 415 p.
5. Letichevsky A., Godlevsky A., Letychevskyy O.(jr.), Potiyenko S., Peschanenko V. Properties of VRS predicate transformer // Cybernetics and System Analysis. – 2010. – Vol. 46. – P. 521–532.

6. Dijkstra E. Guarded commands, nondeterminacy and formal derivation of programs // Communications of the ACM. – 1975. – Vol. 18. – № 8. – P. 453–457.
7. ITU-T Recommendation Z.151 . User requirements notation (URN), 10/2012.
8. Guba A., Shushpanov K. Insertion semantics of flat multithreaded models of UCM // USIM. – 2012. –№6. – P.15–22.
9. Vale M. The evolving algebra semantics of Cobol. Part 1: programs and control. Technical Report CSE-TR-162-93, EECS Dept., University of Michigan. – 1993. – 29 P.
10. Kolchin A., Letychevskyy A., Potiyenko S. A static method for elimination of redundant dependencies in preconditions of transitions of formal models of transition systems. – 2015. – N 1–2. – P. 127–136.
11. Kolchin A. V. An automatic method for the dynamic construction of abstractions of states of a formal model // Cybernetics and system analysis. - 2010. – № 4. – P. 70–90.
12. Kolchin A. V. A method for reduction of analyzed behavior space during verification of formal models of distributed software systems // Artificial intelligence. – 2013. – № 4. – P. 113–126.
13. Kolchin A.V., Kotlyarov V.P., Drobintsev P.D. Method of test scenario generation in insertion modeling environment // Control systems and machines. – 2012. – N 6. – P. 43–48, 63.
14. Drobintsev P., Kolchin A., Kotlyarov V., Letichevsky A., Peschanenko V. An approach to creating concretized test scenarios within test automation technology for industrial software projects // Automatic Control and Computer Sciences. – 2013. – Vol. 47(7). – P. 433–442.

Об авторах:

Губа Антон Андреевич,
кандидат физико-математических наук, младший научный сотрудник.
Количество научных публикаций в украинских изданиях – 11.
Количество научных публикаций в иностранных изданиях – 2.
http://orcid.org/0000-0003-0482-5678,

Колчин Александр Валентинович,
кандидат физико-математических наук, старший научный сотрудник.
Количество научных публикаций в украинских изданиях – 24.
Количество научных публикаций в иностранных изданиях – 11.
http://orcid.org/0000-0001-7809-536X,

Потиенко Степан Валериевич,
кандидат физико-математических наук, старший научный сотрудник.
Количество научных публикаций в украинских изданиях – 15.
Количество научных публикаций в иностранных изданиях – 5.
http://orcid.org/0000-0001-9462-599X.

Место работы авторов:

Институт кибернетики имени В.М. Глушкова НАН Украины.
03680, Киев, проспект Академика Глушкова, 40.
Тел.: (044) 200 8422,
(050) 358 1556.
E-mail: antonguba@gmail.com,
kolchin_av@yahoo.com,
stepan@iss.org.ua.