МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МАТЕРИАЛЫ
VIII Международной молодежной научной
конференции
«МАТЕМАТИЧЕСКОЕ
И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
ИНФОРМАЦИОННЫХ,
ТЕХНИЧЕСКИХ
И ЭКОНОМИЧЕСКИХ СИСТЕМ»

Томск, 26–30 мая 2021 г.

Под общей редакцией И.С. Шмырина

Томск
Издательство Томского государственного университета
2021

Рис. 6. Планировщик задач на сервере

Заключение

Таким образом, поставленная задача по созданию программы, которая поддерживает в актуальном состоянии информацию на стене сообщества социальной сети «ВКонтакте» путем удаления публикаций по таймингу, была выполнена. Благодаря предложенному инструментарию API «ВКонтакте», рассмотренному в работе, предоставляется возможность автоматизации взаимодействия со страницей сообщества, что значительно облегчает создание приложений, предназначенных для управления событиями в рамках данной социальной сети.

В свою очередь использование бота, подобного реализованному, значительно облегчает администрирование сообществ, т.к. избавляет пользователя от необходимости контроля расписания и удаления публикаций вручную.

ЛИТЕРАТУРА

1. Документация API «ВКонтакте» [Электронный ресурс] // VK.COM: социальная сеть «ВКонтакте». URL: <https://vk.com/dev/manuals>
2. Работа с данными в формате JSON [Электронный ресурс] // INTUIT.RU: Национальный Открытый Университет «ИНТУИТ». URL: <https://intuit.ru/studies/courses/19522/1309/lecture/28910>
3. Руководство по PHP [Электронный ресурс] // PHP.NET: сообщество PHP-программистов. URL: <https://www.php.net/manual/ru/index.php>

DOI: 10.17223/978-5-907442-42-9-2021-40

ПРИМЕНЕНИЕ 2-SAT ЗАДАЧИ ДЛЯ ОБНАРУЖЕНИЯ УТЕЧКИ РЕСУРСОВ В СПИСОЧНЫХ СТРУКТУРАХ

Серебренникова К.А., Андреева В.В.

Томский государственный университет

krisert@mail.ru, avv.21@mail.ru

Введение

Тестирование программного обеспечения (далее – ПО) является неотъемлемой частью жизненного цикла разработки ПО. Верификация ПО является одним из важнейших, и, в то же время, трудозатратных этапов разработки. Ошибки, допущенные на стадии верификации ПО, могут привести к существенным потерям в будущем.

Разработка программного обеспечения на языках Си/C++ дает широкие возможности для управления памятью в процессе работы программы. В частности, хранение и обработка данных в списочных структурах позволяет динамически размещать элементы в памяти по мере их необходимости. Здесь самое главное – не допустить утечку ресурсов, т.е. доступ к данным. Потеря доступа приводит к тому, что становится невозможным освобождение памяти из-под таких элементов, что, в свою очередь, приводит

к исчерпанию динамической памяти и аварийному останову программы. Обнаружение таких ситуаций невозможно на уровне компиляции программы, т.к. они являются логическим дефектом и дают о себе знать исключительно в момент работы. Обнаружение такого вида дефектов программы возможно с помощью специальных средств, таких как статические анализаторы. Статические анализаторы позволяют обнаруживать обширное количество ошибок в коде без непосредственного запуска программы, но, к сожалению, статические анализаторы не способны обнаруживать утечки ресурсов, порождаемых неверными действиями пользователей при обработке данных в списочных структурах.

Данная работа посвящена разработке подхода статического анализа, позволяющего обнаруживать такие дефекты. Прилагаемый подход статического анализа кода анализирует именно тот участок, в котором непосредственно выполняются операции над списочными структурами. Сами операции моделируются с помощью рассматриваемой модели 2-SAT, которая представляет логическую формулу вида КНФ, состоящей из дизъюнктов только от двух переменных. С помощью дизъюнктов описываются отношения между элементами динамических структур. Результатом анализа является проверка такой формулы на выполнимость с помощью SAT-решателя. Невыполнимая формула демонстрирует факт отсутствия утечек, т.к. построена модель, в которой нет разрывов между связями элементов.

1. Построение среза кода в виде синтаксического дерева

С целью реализации статического анализа исходный код программы представляется в виде абстрактного синтаксического дерева, или, другими словами, преобразуется в структуру, удобную для дальнейшего анализа (от англ. AST – abstract syntax tree). AST в информатике – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами [1]. Иными словами, это исходный код программы, представленный в виде структуры, удобной для дальнейшего анализа. В листингах 1 и 2 продемонстрированы код метода **PushBack**, который добавляет элемент в конец списка, и код в виде синтаксического дерева.

<pre>void PushBack(Node* head, int value) { Node*last = GetLast(head); Node* newNode (Node*)malloc(sizeof(Node)); newNode->value = value; if (last == NULL) { newNode->next = newNode; } else{ last->next = newNode; } }</pre>	<pre>PushBack(method):in "head"<Node*>, in "value"<int> ["last"<Node*>: { value: getLast("head") next : getLast("head") }; if ("last" == NULL) { "head"<Node*> { next:newNode<Node*> }; } else { "last" <Node*>: { next:newNode<Node*> }; }]</pre>

Листинг 1. Исходный код метода Push Back

Листинг 2. Код метода Push Back, представленный в виде синтаксического дерева

В качестве метода построения среза кода с последующим представлением его в виде синтаксического анализа воспользуемся подходом, предложенным в [2]. Данный подход позволяет выделить подмножество кода, необходимое для анализа, в частности, именно те участки, которые относятся обработке списочной структуры.

2. Построение графов списочных структур

Метод анализа списочных структур базируется на последовательном анализе синтаксического дерева и формировании для каждого блока кода, связанного с выполнением некоторых действий над элементами структуры, особого графа G – графа списочных структур. Граф G состоит из множества вершин, являющихся элементами списочной структуры, ребра соответствуют связям между элементами. Здесь необходимо заметить, что при работе со списочными структурами элементы создаются в динамической памяти (куче), причем для организации корректной работы обязательно нужно сохранять адреса этих объектов. В случае, например, таких структур, как линейные списки, достаточно хранить адрес на первый элемент. Следовательно, мы можем выделить два типа вершин: вершины, соответствующие переменным, хранящим адреса объектов и объекты, непосредственно хранящиеся в динамической памяти.

Пусть $V = \{V_i\}$ – множество переменных, хранящих адреса элементов динамической структуры, $N = \{N_i\}$ – множество адресов объектов, хранящихся в динамической памяти.

Построение графа выполняется в результате анализа операций, проводимых над списками. В частности, создание элементов списочной структуры, встраивание их в структуру порождает формирование узлов графа и проведение ребер между вершинами, соответствующими связям между элементами в списочной структуре. Рассмотрим построение графа на конкретном примере.

На рис. 1 представлен граф после создания «головного» указателя типа $Node^* head$, где V_1 – переменная, хранящая адреса элементов динамической структуры. Т.к. в списке отсутствуют элементы, то «головной» указатель ссылается на NULL.

Рис. 1. Граф после создания «головного» элемента

Далее на рис. 2 и 3 продемонстрированы графы после добавления элементов с помощью метода *PushBack* (Листинг 1). Рассмотрим, каким образом элементы связываются между собой в списочную структуру данных. Связь элементов происходит с помощью указателя *next* (поле структуры *Node*). При добавлении элемента в «хвостовую» часть списка могут возникнуть 2 случая:

1. Изначально список пустой. В данном случае для корректной работы программы необходимо переопределить указатель *head* на новый элемент, а новый элемент направить в NULL.

2. Изначально список непустой. Если в списочной структуре уже содержатся элементы, то необходимо перенаправить предыдущий последний элемент на новый, а новый направить в NULL.

Рис. 2. Граф после добавления первого элемента

Рис. 3. Граф после добавления второго элемента

3. Формирование логической формулы для графов списочных структур в виде КНФ, проверка на выполнимость, формулировка выводов

Рассмотрим логическую модель, описанную в [3], которую мы можем применить к проблеме утечки памяти в списочных структурах данных. Для поиска утечки необходимо замкнуть последний элемент с «головным». Таким образом, проблема поиска утечек памяти сводится к проверке связности графа. Связный граф – граф, между любой парой вершин которого существует как минимум один путь [4].

Как показано в [3], ориентированный граф можно представить с помощью логической формулы КНФ, состоящей из дизъюнктов от двух переменных. Эти переменные советуют вершинами графа. Переменная, из которой выходит ребро, входит в дизъюнкт без знака инверсии, переменная в которую входит ребро, входит в дизъюнкт со знаком инверсии. Воспользуемся выводами из [3]: граф является связным тогда и только тогда, когда задача 2-SAT имеет ровно два решения: первое – это то, где каждая логическая переменная истинна (что называется присвоением белого), второе – то, где каждая логическая переменная ложная (что называется присвоением черного). Такую задачу авторы называют «черно-белой» SAT-задачей.

Следовательно, при добавлении отрицания «белого» и «черного» решений к сгенерированной задаче SAT она становится невыполнимой.

Замечание 1. Если вершина x_1 имеет ребра к вершинам x_2 и x_3 , то логическая модель будет следующей: $(x_1 \rightarrow x_2) \wedge (x_1 \rightarrow x_3)$.

Замечание 2. Формула $(x_1 \rightarrow x_2) \wedge (x_1 \rightarrow x_3)$ преобразовывается в задачу 2-SAT с помощью следующего правила: $x \rightarrow y = x \vee \bar{y}$.

Для поиска решения используется SAT-решатель CryptoMiniSat [5]. Применим данную модель к задаче поиска утечек памяти в линейном списке.

Т.к. модель предполагает проверку связности графа, необходимо замкнуть следующим образом: последний элемент списка, направленный в NULL, замыкаем с «головным». Получаем граф, изображенный на рис. 6.

Рис. 6. Граф линейного списка: V_1 – указатель head на первый элемент списка, $N_1 - N_n$ – узлы (элементы) списка

Т.к. ребра графа отражают связи между элементами списка, при отсутствии хотя бы одного ребра между вершинами графа, он становится несвязным.

Перепишем граф в виде формул: $V_1 \rightarrow N_1$, $N_1 \rightarrow N_2$, $N_2 \rightarrow N_3$, ..., $N_{n-1} \rightarrow N_n$, $N_n \rightarrow V_1$. Составим КНФ для SAT-решателя, используя замечание 2:

$$(\bar{V}_1 \vee N_1) \wedge (\bar{N}_1 \vee N_2) \wedge (\bar{N}_2 \vee N_3) \wedge \dots \wedge (\bar{N}_{n-1} \vee N_n) \wedge (\bar{N}_n \vee V_1). \quad (1)$$

Согласно [2], если граф связный, т.е. отсутствуют разрывы между элементами списка, формула (1) имеет 2 решения («черное» и «белое»). Добавим в формулу (1) «белое» и «черное» решения в виде двух последних конъюнктов:

$$(\bar{V}_1 \vee N_1) \wedge (\bar{N}_1 \vee N_2) \wedge (\bar{N}_2 \vee N_3) \wedge \dots \wedge (\bar{N}_{n-1} \vee N_n) \wedge (\bar{N}_n \vee V_1) \wedge \\ \wedge (V_1 \vee N_1 \vee N_2 \vee N_3 \vee \dots \vee N_n) \wedge (\bar{V}_1 \vee \bar{N}_1 \vee \bar{N}_2 \vee \bar{N}_3 \vee \dots \vee \bar{N}_n).$$

Таким образом, опираясь на описанную выше модель, сделаем выводы о наличии/отсутствии утечек памяти в программном коде:

– Если полученная формула невыполнима, то граф связный. Из этого можно сделать вывод, что разрыва связей между элементами нет и соответственно утечки памяти не обнаружены.

Если формула выполнима, то граф несвязный, т.е. утечка памяти обнаружена.

Сформулируем алгоритм поиска утечек памяти в списочных структурах данных.

Алгоритм состоит из следующих этапов

1. Построение среза кода в виде синтаксического дерева.
2. Построение графов списочных структур.
3. Формирование логической формулы для графов списочных структур в виде КНФ.
4. Анализатор на основе SAT-решателя.

Предложенный алгоритм программно реализован. Тестирование выполнялось на специально подготовленных примерах, включающих в себя многочисленные варианты утечек. В частности, рассматривались все операции, приводящие к утечкам ресурсов. К таким операциям относятся, сдвиг указателя, переопределение указателя и неверная установка связей между элементами.

Заключение

В данной работе основной акцент сделан на поиск утечек памяти в динамических структурах списочного типа и основное внимание уделено получению возможности обнаруживать нарушение связей между элементами списочных структур. В результате, был разработан алгоритм, который упрощает поиск утечек памяти в программах, использующих динамические структуры списочного типа.

ЛИТЕРАТУРА

1. Абстрактное синтаксическое дерево [Электронный ресурс] URL: Абстрактное синтаксическое дерево – Википедия (wikipedia.org); (дата обращения 01.05.2021).
2. *Takahashi M., Mizukoshi N., Tsuda K.* Efficient Program Verification Using Binary Trees and Program Slicing // In: Negoita M.G., Howlett R.J., Jain L.C. (eds) Knowledge-Based Intelligent Information and Engineering Systems. KES 2004. – Lecture Notes in Computer Science. Springer, Berlin, Heidelberg. – 2004. – V. 3213. – https://doi.org/10.1007/978-3-540-30132-5_69
3. *Biro C., Kasper G.* Equivalence of strongly connected graphs and black-and-white 2-SAT problems // Miskolc Mathematical Notes. – 2018.
4. Связный граф [Электронный ресурс] URL: https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D0%B3%D1%80%D0%B0%D1%84; (дата обращения 20.04.2021).
5. CryptoMiniSAT [Электронный ресурс] URL: https://msoos.github.io/cryptominisat_web/#/; (дата обращения 03.05.2021).