DOI 10.24412/1829-0450-fm-2026-1-36-43
УДК 519.688

Поступила: 09.03.2026г.
Сдана на рецензию: 10.03.2026г.
Подписана к печати: 19.03.2026г.

ОПРЕДЕЛЕНИЕ ВЕРСИИ КОМПИЛЯТОРА SOLIDITY ИЗ ИСХОДНОГО КОДА

Т.С. Авагян, О.М. Мовсисян, А.К. Асланян

Российско-Армянский университет
tigran.avagyan@rau.am, hovhannes.movsisyan@rau.am, hayk.aslanyan@rau.am
ORCID: 0009-0007-0924-641X, 0000-0002-7582-7948, 0000-0002-7320-4835

АННОТАЦИЯ

В данной работе представлен метод и реализован инструмент для нахождения версии компилятора из исходного кода для языка “Solidity”. Подход реализован на основе регулярных выражений, который обеспечивает производительность в сравнении с другими методами работающими на базе абстрактного синтаксического дерева. Экспериментальная оценка на реальных проектах продемонстрировала его эффективность. Производительность разработанного инструмента в 6 раз быстрее существующего альтернативного решения; точность достигает 100% (против 88,41%), а полнота – 98,77% (против 79,04%).

Ключевые слова: блокчейн, анализ кода, компиляторы.

Введение

Языки программирования подвергаются множественным модификациям, прежде чем достигнут своего окончательного вида. Дело в том, что, чаще всего, сами разработчики не уверены, каким должен быть окончательный язык: его синтаксис, детали имплементации и т.д. Язык может измениться очень радикально от версии к версии, поэтому возникает сильная необходимость компилирования именно конкретной версией компилятора. Поэтому при ранних версиях языка принято как-то обозначать версию, на котором надо будет компилировать программу.

Одним из таких языков является основной язык для разработки смарт-контрактов [1] на блокчейне “Ethereum” [2] – “Solidity” [3]. Версия языка может обозначаться как конкретно, так и интервалами, что на первый взгляд упрощает нахождение версии, давая широкий выбор, но на самом деле – это не так. Проблема в том, что интервалов может быть много (н.п. pragma solidity >0.8.0 <0.9.0 >=0.8.7), и выбранная версия компилятора должна находиться в каждом из них.

Простейшим решением может быть перебор всех версий и компиляция файла с каждым из них. Если будет ошибка компиляции, мы пропустим эту

версию, иначе – мы отметим, что эта версия удовлетворяет требованиям. Основная проблема этого решения – производительность. При компиляции больших файлах компиляция может длится слишком долго.

Постановка задачи

Цель: разработать систему, которая находит все версии компилятора для данного файла, соответствующие всем требованиям.

Для достижения цели были выполнены следующие задачи:

1. Сделать предобработку кода, облегчив нахождение *специальных инструкций*.
2. Найти в коде специальные инструкции, ограничивающие версию компилятора.
3. Парсить найденные инструкции и найти ограничения в *канонической форме*.
4. Найти пересечение всех ограничений.

В постановке задачи использовались выражения *специальная инструкция* и *каноническая форма*.

В “Solidity” версия компилятора задается с помощью директив «version pragma» [3]. Это инструкции, которые могут в себе содержать как саму версию, так и некоторый интервал, либо несколько интервалов. Их в коде может быть несколько, и они могут находиться как в начале кода (как принято), так и в середине либо в конце. Их будем называть *специальными инструкциями*.

Множество всех версий “Solidity” обозначим через **V**. *Канонической формой* будем называть специальную форму представления множества видов: >v, >=v, <v, <=v, =v, ^v, ~v где v – версия “Solidity”, которую можно представить в виде вектора (v1, v2, v3) множества $\mathbb{N}_0^3$. Для версий можем определить отношение порядка < данным образом:

$$v < u <=> \begin{cases} v1 < u1 \\ v1 = u1 \wedge v2 < u2 \\ v1 = u1 \wedge v2 = u2 \ \wedge v3 < u3 \end{cases}$$

Ниже перечислены формальные представления множеств заданных канонической формой:

$\forall v \in V$

$>v \equiv \{u \in V \mid u > v\}$

$>=v \equiv \{u \in V \mid u \geq v\}$

$<v \equiv \{u \in V \mid u < v\}$

$<=v \equiv \{u \in V \mid u \leq v\}$

$=v \equiv \{v\}$

$\wedge v \equiv \{u \in V \mid u1 = v1 \wedge u2 = v2 \wedge u3 \geq v3\}$

$$\sim_{\mathbf{V}} \equiv \quad \wedge_{\mathbf{V}}$$

(так как для всех элементов V первая компонента ровна нулю).

Предобработка

Ввод: код на языке “Solidity”;

Вывод: предобработанный код.

Разработанный алгоритм работает на основе регулярных выражений [4], однако, если специальная инструкция будет закомментирована, мы не сможем различить ее от незакомментированные специальной инструкции. Появляется необходимость удалить все комментарии из кода перед основным алгоритмом. В этом и заключается предобработка.

Комментарии в “Solidity” бывают двух видов: однострочные и многострочные. Однострочные комментарии начинаются на // и заканчиваются, как только заканчивается строка. Многострочные комментарии начинаются с /* и заканчиваются на */, при этом каждый символ не может находится одновременно в однострочном и многострочном комментариях.

В реализации алгоритма алгоритм обходит все символы в коде, игнорируя содержимое строковых литералов, если он еще не находится в многострочном комментарии. Если алгоритм вошел в многострочный комментарий, он игнорирует все символы, пока не выйдет из него. Если он вошел в однострочный комментарий, он сразу переходит на новую строку.

В итоге, у нас есть код, в котором удалены все комментарии.

Нахождение специальных инструкций

Ввод: предобработанный код;

Вывод: специальные инструкции.

Специальные инструкции имеют вид

pragma solidity L,

где L – выражение, вида k1, k2, …, kn, где ki – множество, заданное в *каноническом виде*. Сложность нахождения *специальных инструкций* заключается в том, что *специальные инструкции* могут встречаться в любом месте в коде, и надо учитывать все специальные инструкции. Более того, выражение может не быть именно в таком формате, имея в виду, что между токенами могут быть пробелы, горизонтальные табуляции и переводы строки.

Как было отмечено выше, алгоритм реализован с помощью регулярных выражений.

Поиск регулярного выражения осуществлялся итеративно, на каждом шаге усложняя выражение, тем самым приводя его к нужному нам выражению.

Разделим *специальную инструкцию* на три части: *пролог, главная часть* и *эпилог*. *Пролог специальной инструкции* – это часть, в котором присутствует только фрагмент pragma solidity

Имея ввиду все пробелы, горизонтальные табуляции и символы перевода строки построим регулярное выражение только для этой части.

^\s*pragma\s+solidity\s*

Эпилог состоит из одного символа `;`. Она нужна, чтобы компилятор знал, когда заканчивается инструкция. Для него регулярное выражение очевидно.

Главной частью является часть содержащая версии. Рассмотрим упрощенный вариант *главной части*, когда во фрагменте одна *каноническая форма*. Между оператором и операндом могут быть пробелы и другие похожие символы, версия может быть задана как с помощью трех чисел рассмотренных ранее, так и с помощью двух чисел, опуская последний 0. Принимая в расчет то, что между точками могут быть пробелы и похожие символы, в результате получаем регулярное выражение следующего вида:

(\^|>|>=|<|<=|=|~)?[\s?]*\d+\s*\.\s*\d+(\s*\.\s*\d+)?\s*

Так как в *главной части* количество *канонических форм* строго больше нуля, получим:

((\^|>|>=|<|<=|=|~)?[\s?]*\d+\s*\.\s*\d+(\s*\.\s*\d+)?\s*)+

Также, *главная часть* может быть заключена в " или в '. Так как нахождение версии имеет смысл только с файлами, которые компилируются, мы можем, вместо того, чтобы проверить для каждого типа кавычек, проверить начало и конец на наличие кавычек любого типа.

["']?(((\^|>|>=|<|<=|=|~)?[\s?]*\d+\s*\.\s*\d+(\s*\.\s*\d+)?\s*)+)["']?

Если кавычки не будут соответствовать друг другу, ни с какой версией компилятора код не скомпилируется, поэтому тот код будет нам не интересен. В итоге получили регулярное выражение, вида:

^\s*pragma\s+solidity\s*["']?(((\^|>|>=|<|<=|=|~)?[\s?]*\d+\s*\.\s*\d+(\s*\.\s*\d+)?\s*)+)["']?;

Данное регулярное выражение позволяет находить все специальные инструкции.

Парсинг специальных инструкций

Ввод: *специальные инструкции*;

Вывод: *канонические формы*.

Имея все *специальные инструкции*, можно извлечь главные части. В итоге, мы имеем главные части всех *специальных инструкций*. С помощью регулярного выражения

(\^|>|>=|<|<=|=|~)?[\s?]*(\d+)\s*\.\s*(\d+)(\s*\.\s*(\d+))?

получим ограничения в следующем виде

(оператор, (v1, v2, v3)) ∈ {^, >, >=, <, <=, =, ~} × $\mathbb{N}_0^3$

Если (v1, v2, v3) обозначим через v, а оператор присоединим к v слева, получим *каноническую форму*.

Пересечение канонических форм

Ввод: *канонические формы*;

Вывод: множество версий, удовлетворяющие требованиям всех *канонических форм*

Так как множество **V** невелико (менее ста элементов), мы можем сделать перебор по всем его элементам и проверить, находится ли элемент в пересечении множеств данных каноническими формами. Причина представления множеств в *канонической форме* удобно тем, что мы можем не хранить эти множества как набор элементов, а хранить как пару (оператор, версия), что будет занимать значительно меньше памяти. Возникает вопрос, как именно проверить, находится ли какая-то версия в множестве, заданной *канонической формой*. Поскольку канонические формы определяются с помощью неравенств, для заданной версии достаточно проверить условия, определяющие форму, чтобы установить принадлежность к множеству, индуцированному данной формой. После перебора всех версий из **V**, как результат, получим множество версий, удовлетворяющие требованиям всех *канонических форм*.

Оптимизации

Если допустить, что существует хотя бы одна версия, при которой файл успешно компилируется, можно применить следующую оптимизацию. Можно искать в канонических формах множество, вида =v либо, если сокращенно: v (если такой имеется), можно сразу же закончить поиск, дав в качестве результата версию v.

Для следующей оптимизации, надо тщательнее исследовать каноническую форму. Обозначим через $\mathbf{V}_<$ отсортированный набор, содержащий все элементы множества **V**. Пронумеруем элементы набора $\mathbf{V}_<$.

$\mathbf{V}_< = (v^1, v^2, ..., v^n)$

Рассмотрим *форму* вида =v (v).

=v = {v} = (<=v) ∩ (>=v)

Заменим все *формы* вида ~v на ^v. Рассмотрим *форму* вида ^v. По определению:

^v = {u ∈ **V** | u1 = v1 ∧ u2 = v2 ∧ u3 > v3} =

= {u ∈ **V** | u ≥ v} ∩ {u ∈ **V** | u < (v1+1, 0, 0)}

v' ≡ (v1+1, 0, 0)

^v = {u ∈ **V** | u ≥ v} ∩ {u ∈ **V** | u < v'} = (>=v) ∩ (<v')

Если v' ∉ **V**, то <v' = **V**.

>=v ⊂ **V** ⇒ (>=v) ∩ V = (>=v) ⇒ ^v = (>=v)

Если v' ∈ **V**, то ^v = (>=v) ∩ (<v').

В итоге, мы разложили *формы* ^v и =v при помощи остальных *форм*. Все *формы* вида ^v заменим на соответствующие ему эквивалентные *формы*. При

первом случае ^v заменяется на >=v, во втором же случае заменим ^v на две *формы*, которые мы все равно пересечем в последней фазе. Аналогично поступим и с *формами* вида =v. Мы можем и дальше упрощать *формы* и прийти только к *формам* вида >=v и <=v, но в этом нет необходимости.

В результате, получаются только *формы*, видов >=v, >v, <=v, <v. Первые два назовем *каноническими формами типа 1* (*к1*), а последние два – *каноническими формами типа 2* (*к2*). Добавим к *каноническим формам формы* $>=v^1$ и $<=v^n$. Они никак не изменят результат пересечения. Теперь мы имеем хотя бы одну *к1* и хотя бы одну *к2*. Пересечение всех *канонических форм* можно вычислить, взяв пересечение всех *к1* и всех *к2* и вычислить пересечение полученных двух множеств. При пересечении двух *к1*, получим *к1*, и при пересечении двух *к2*, получим *к2*. Обобщив результат, пересечение всех *к1* дает *к1*, а пересечение всех *к2* дает *к2*. Таким образом, для нахождения результата надо найти пересечение одной *к1* с одной *к2*, что даст либо пустое множество, либо множество с одним элементом, либо множество элементов, которое мы отсортируем и полученный набор обозначим через R.

$\exists i\ \exists j\ R = (v^i, v^{i+1}, ..., v^j)$

Получили, что для результата, нужно найти i и j. Для нахождения i можно перебирать элементы $\mathbf{V}_<$ начиная слева. Первая версия, удовлетворяющая всем условиям будет версией под номером i. Затем сделаем бинарный поиск для нахождения j следующим образом: обозначим i через l0, а n через r0. Возьмем середину отрезка [l0; r0], обозначим через k1 и проверим, удовлетворяет ли она всем требованиям специальных инструкций. Если удовлетворяет, возьмем в качестве l1 число k1, а в качестве r1 число r0. Если не удовлетворяет, возьмем в качестве l1 число l0, а в качестве r1 число k1. Сделаем то же самое для отрезка [l1; r1]. В итоге, после конечного числа шагов, получим lp и rp числа, которые равны между собой.

lp = rp = j

Результаты

Разработанный инструмент был тестирован на наборе из 895 смарт-контрактов на языке “Solidity”. Для сравнения мы взяли инструмент solcix [6], который дает возможность решить нашу проблему. Результаты представлены в Табл. 1. В первом столбце написаны имена инструментов (первый инструмент – наш), второй столбец показывает количество найденных и проверенных версий, третий столбец показывает количество файлов, для которых инструмент выдал ошибку при анализе, а последний столбец показывает время, потраченное на анализ всех файлов. Наш инструмент работает на 6.3 раза быстрее, так как работает на основе регулярных выражений. Для 11-и файлов никакой из инструментов не смог найти версию.

Таблица 1. Результаты сравнения.

Инструмент	Количество правильно найденных версий	Количество ложно-положительных результатов	Время
SolVerScan	884	0	17.8 с.
solcix	641	84	112.0 с.

Заключение

В данной статье предложен и реализован метод определения версии компилятора “Solidity” по исходному коду. Показано, что использование регулярных выражений позволяет существенно повысить производительность по сравнению с подходами, основанными на анализе абстрактного синтаксического дерева.

Экспериментальная оценка на реальных проектах подтвердила высокую точность и быстродействие предложенного метода. Реализация инструмента доступна в открытом доступе под лицензией MIT [7], что делает его применимым для практического использования и дальнейших исследований.

ЛИТЕРАТУРА

1. *Сабо Н.* Smart Contracts: Building Blocks for Digital Markets // “Extropy”. 1996. № 16. PP. 18–23.
2. *Хилденбрандт Е., Саксена М., Чжу С., Родригес Н., Дациан П., Гут Д., Мур Б., Чжан И., Парк Д., Штефанеску А., Рошу Г.* KEVM: A Complete Semantics of the Ethereum Virtual Machine // Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF 2018). IEEE Computer Society, 2018. CC. 204–217. DOI: 10.1109/CSF.2018.00022.
3. *Закжревский Я.* Towards Verification of Ethereum Smart Contracts: A Formalization of Core of Solidity / Ред. Р. Пискач, П. Руммер. В сборнике Verified Software. Theories, Tools and Experiments: 10th Int’l Conf., VSTTE 2018, Revised Selected Papers. Сер.: Lecture Notes in Computer Science, т. 11294. Cham: Springer, 2018. CC. 229–247. DOI: 10.1007/978-3-030-03592-1_13.
4. *Кампэану К., Саломаа К., Ю.Ш.* A formal study of practical regular expressions // “International Journal of Foundations of Computer Science”. Т. 14, № 6, 2003. CC. 1007–1018. DOI:10.1142/S012905410300214X.
5. *Ахо А.В., Лэм М.С., Сети Р., Ульман Д.Д.* Compilers: Principles, Techniques, and Tools / 2-е изд. Boston: Addison-Wesley, 2006. CC. 108–120.
6. *Solratic*. solcix: The ultimate Solidity compiler version management tool and Python package / GitHub repository. URL: https://github.com/Solratic/solcix (дата обращения: 14.01.2026).
7. cast-tech. Solverscan / GitHub repository. URL: https://github.com/cast-tech/solverscan (дата обращения: 16.01.2026).

SOLIDITY COMPILER VERSION DETECTION FROM SOURCE CODE

T. Avagyan, H. Movsisyan, H. Aslanyan
Russian-Armenian University

ABSTRACT

This paper presents a method and an implementation of a tool, that finds the compiler version from the source code for the language Solidity. The approach uses regular expressions on its base, which ensures high performance compared to other methods based on the abstract syntax tree. Experimental evaluation on real-life projects has demonstrated its effectiveness, and the comparison with an existing tool has demonstrated its performance and accuracy. The developed tool demonstrates a performance increase of six times over the existing alternative solution; its precision reaches 100% (compared to 88.41%), while its recall is 98.77% (compared to 79.04%).

Keywords: blockchain, code analysis, compilers.