Санкт–Петербургский государственный университет
Факультет математики и компьютерных наук

ГЛАДШТЕЙН Владимир Петрович

Выпускная квалификационная работа

Тема работы: Верификация алгоритма проверки моделей для многопоточных программ в системе доказательств Coq

Уровень образования: бакалавриат
Направление 01.03.01 «Математика и компьютерные науки»
Основная образовательная программа «Математика»

Научный руководитель:
доцент, д.ф.-м.н. Д. Ю. Булычев

Рецензент:
Research Programmer, Zilliqa Research Pte. Ltd., Singapore
А. А. Трунов

Санкт-Петербург
2022 г.

Содержание

Введение . . . 3

Постановка задачи . . . 6

1. Обзорный раздел по предметной области . . . 7

1.1. Графы исполнения . . . 7

1.2. Модели памяти . . . 9

1.3. Алгоритм TruSt . . . 10

1.4. Границы применимости TruSt . . . 14

1.5. Язык Coq . . . 15

1.6. Библиотека Hahn . . . 16

2. Теоремы о Корректности и Терминируемости . . . 18

2.1. TruSt в терминах отношений . . . 18

2.2. Теорема о Корректности . . . 19

2.3. Теорема о Терминируемости . . . 19

3. Теоремы о Полноте и Оптимальности . . . 23

3.1. Алгоритм Prev и его базовые свойства . . . 23

3.2. Теорема о Полноте . . . 27

3.3. Теорема об Оптимальности . . . 28

Заключение . . . 30

Список литературы . . . 31

Введение

Проверка моделей без сохранения состояния (stateless model checking, SMC) [1, 2] — это эффективный метод верификации конечных параллельных программ. Он был представлен как альтернатива проверке модели с явным сохранением состояния, которая позволяет избежать чрезмерного потребления памяти и, следовательно, имеет потенциал для масштабирования до более крупных программ. SMC работает систематически исследуя все сценарии исполнения программы данной параллельной программы, не сохраняя набор уже посещенных состояний программы.

Хотя SMC позволяет проверять программу с полиномиальными затратами по памяти, у него есть очевидный недостаток, заключающийся в том, что количество проверяемых исполнений обычно экспоненциально зависит от размера программы. Таким образом, SMC почти всегда используется вместе с продвинутыми алгоритмами динамической редукции частичного порядка (DPOR) [3, 4, 5], сокращающими количество выполнений, которые необходимо изучить, чтобы охватить все поведения программы. DPOR разделяет трассы исполнения программы на классы эквивалентности по некоторому отношению, такому как эквивалентность Мацуркевича [6], с тем свойством, что все эквивалентные трассы показывают один и тот же наблюдаемый результат. Тогда для верификации программы достаточно исследовать по одной трассе из каждого класса эквивалентности.

Однако достижение оптимального алгоритма (в ходе работы которого, исследуется только одна трасса исполнения из каждого класса эквивалентности) — непростая задача. Алгоритмы DPOR (например, [7, 9, 10, 4, 5, 11]) обычно начинают с изучения одной трассы программы, и всякий раз, когда они обнаруживают пару обращений к памяти, находящихся в состоянии гонки, они исследуют дополнительные трассы, которые содержат такие обращения в обратном порядке, а также сохраняют некоторое состояние, чтобы избежать повторного исследования эквивалентных трасс исполнения. Однако существующие алгоритмы либо неоптимальные, что означает, что они могут исследовать экспоненциальное число трасс даже для программ с $O(n)$ классами эквивалентности (где n — размер программы), либо достигают

оптимальности, сохраняя какие-то исполнения программы и, следовательно, жертвуют тем, для чего SMC был придуман: полиномиальным потреблением памяти. На самом деле, большинство современных решений DPOR для проблемы верификации выбирают второе решение, т. е. они могут потреблять экспоненциальный объем памяти.

Наконец, командой института Макса Планка (Кайзерслаутерн, Германия) был представлен алгоритм TruSt (Truly Stateless Model-checker), который позволяет избежать данного компромисса. Он производит оптимальный DPOR и в тоже время работает с линейными затратами по памяти. Еще одно преимущество TruSt заключается в том, что он параметричен в выборе модели памяти, поддерживая не только модель последовательной согласованности (SC) [12], но и широкий спектр слабых моделей памяти, таких как модель TSO [13], PSO [14] и RC11 [15].

Однако реализация самого алгоритма нетривиальная. Ошибка в такой реализации приведет к тому, что алгоритм либо будет вводить пользователя в заблуждение, либо находить не все ошибки. Поэтому очень важно верифицировать такие алгоритмы.

Обычно доказательства корректности в области верификации многопоточных программ очень громоздкие и нетривиальные. Они требуют рассмотрения большого количества случаев, некоторые из которых доказываются механически. В этом деле человеку сложно учесть все детали и не допустить ошибок. В тоже время, как было описано ранее, выдвигаются повышенные требования к надежности. Чтобы преодолеть данную проблему, можно использовать системы автоматической проверки доказательств, например Coq [16]. Такие инструменты позволяют записывать формулировки различных теорем в качестве кода на соответствующем языке программирования, а затем в рамках того же языка строить доказательства. После система проверяет предъявленные доказательства на наличие ошибок. Подобный процесс взаимодействия с подобными системами называется механизацией доказательств. Механизация как раз и гарантирует корректность представленного кода. Системы по типу Coq очень хорошо показали себя в значительном количестве больших проектов. В частности, они были использованы для написания верифицированного компилятора языка C [27], ядра ОС [28], файловой системы [30], а так же

для проверки доказательств теорем “Фейхта-Томпсона” [31] и “О четырех красках” [29].

Целью данной работы было получение механизированного доказательства корректности работы алгоритма TruSt используя систему Coq.

Постановка задачи

Целью данной работы была спецификация и формальная верификация алгоритма конечной проверки моделей TruSt с использованием системы автоматической проверки доказательств Coq. Для достижения этой цели были поставлены следующие задачи.

- Доказать теорему о корректности алгоритма TruSt: все сценарии исполнения программы, посещаемые в ходе работы алгоритма, корректны относительно заранее выбранной модели памяти.
- Доказать теорему о терминируемости алгоритма TruSt: алгоритм терминируется на любой конечной программе.
- Доказать теорему о полноте алгоритма TruSt: любое исполнение конечной программы, удовлетворяющее выбранной модели памяти, будет достигнуто в ходе исполнения алгоритма.
- Доказать теорему об оптимальности алгоритма TruSt: алгоритм посещает один класс эквивалентности исполнений программы не более одно раза.

1. Обзорный раздел по предметной области

В этой главе будут рассмотрены понятия, на которых основывается работа. В частности, будут введены графы исполнения, как способ кодировки сценариев исполнения программ и будет введен алгоритм TruSt. Также будут определены границы применимости TruSt, которые необходимы для дальнейших доказательств и рассмотрены используемые инструменты.

1.1. Графы исполнения

Как уже было написано ранее, алгоритмы SMC в ходе своей работы исследуют различные сценарии исполнения. Ясно, что для этого такие сценарии должны быть закодированы с помощью некоторой структуры данных. Если необходимо поддержать DPOR, то такая структура должна кодировать сразу несколько эквивалентных, с точки зрения наблюдаемого эффекта, сценариев исполнения. В рамках данной работы будут использованы графы исполнения. Такие графы содержат в себе множество событий и некоторые отношения на них.

Далее под $\mathtt{Loc}$ будем понимать множество локаций (адресов) разделяемых переменных, под $\mathtt{Tid}$ — множество идентификаторов потоков, под $\mathtt{Lab}$ — множество меток. Последнее множество стоит из следующих элементов.

- Метка чтения $\mathtt{R}(l)$, где $l \in \mathtt{Loc}$ — локация, из которой происходит чтение
- Метка записи $\mathtt{W}(l, v)$, где $l \in \mathtt{Loc}$ — локация, в которую происходит запись значения $v \in \mathtt{Val}$
- Метка ошибки $\mathtt{Error}$

Определение 1 (Множество событий)**.** *Множество* $\mathtt{Event}$ *называется множеством событий если для любого* события $e \in \mathtt{Event}$*, либо*

- $e = \langle \mathtt{init}\ l \rangle \in \mathtt{Event}_0 \subseteq \mathtt{Event}$*, для некоторого* l *в множестве локаций* $\mathtt{Loc}$*, то есть* e *является событием инициализации, либо*

- *$e = \langle t, i, lab \rangle$, для t из множества идентификаторов потоков* $\texttt{Tid}$*, i — порядкового номера внутри одного потока из $\mathbb{N}$ и lab из множества меток* $\texttt{Lab}$*, то есть e является событием потока.*

Когда применимо, функции $\texttt{tid}$, $\texttt{idx}$, $\texttt{loc}$ и $\texttt{val}$ возвращают идентификатор потока, порядковый номер, локацию и значение события соответственно. Будем использовать $\texttt{R}$, $\texttt{W}$ и $\texttt{Error}$ для обозначения событий соответствующего типа, и используем нижние индексы для дальнейшего ограничения этих наборов (например, $\texttt{W}_l = \{w \in \texttt{W} \mid \texttt{loc}(w) = l\}$).

Определение 2 (Граф исполнения)**.** *Граф исполнения — это граф, включающий в себя*

- *Последовательность вершин $G.\texttt{E}$, такую что $\langle \texttt{init}\ l \rangle \in G.\texttt{E}$, для всех $l \in \texttt{Loc}$*

- *Так называемую* reads-from *функцию $G.\texttt{rf} : G.\texttt{R} \rightarrow G.\texttt{W}$, которая отправляет каждое чтение в соответствующую ему запись в ту же локацию. Если $G.\texttt{rf}(r) = w$, говорят что чтение r читает из записи w*

- *Отношение когерентности $G.\texttt{co} \subseteq \bigcup_{l \in \texttt{Loc}} G.\texttt{W}_l \times G.\texttt{W}_l$, которое является строгим частичным порядком. Кроме того, это отношение должно быть тотальным на множестве записей в одну локацию*

Далее для множества X, $G.X$ обозначает $G.\texttt{E} \cap X$. Так же для $e_1, e_2 \in G.\texttt{E}$ будем писать $e_1 <_G e_2$, если e_1 встречается в $G.\texttt{E}$ раньше чем e_2. $G|_X$ обозначает сужение графа G на множество X, а $G \backslash X := G|_{G.\texttt{E} \backslash X}$. Кроме того будем писать $G \sim G'$ если графы G и G' равны с точностью до перенумеровки их событий.

Наконец, можно определить отношение программного порядка $\texttt{po}$ на событиях графа. Это отношение является отношением частичного порядка, упорядочивающим события внутри одного потока. Кроме того, любые начальные состояния являются минимальными по отношению к нему.

Определение 3 (Отношение программного порядка $\texttt{po}$)**.**

$$\texttt{po} := \texttt{Event}_0 \times (\texttt{Event} \backslash \texttt{Event}_0) \cup \{(\langle t, i_1, l_1 \rangle, \langle t, i_2, l_2 \rangle) \mid i_1 < i_2\}$$

Для любого отношения R, за $G.R$ далее будем обозначать отношение R суженное на $G.\mathtt{E} \times G.\mathtt{E}$. Кроме того введем следующее отношение причинно-следственной связи.

$$G.\mathtt{porf} := (G.\mathtt{po} \cup G.\mathtt{rf})^{+}$$

Для примера, рассмотрим программу

$$(1)x := 1; \;\Big\|\; (2)y := 1; \;\Big\|\; \begin{array}{l}(3)a := x;\\(4)b := y\end{array} \qquad (1)$$

Тут || обозначает параллельную композицию потоков. Этой программе соответствуют множество различных графов исполнения. Например те, которые изображены на рисунке 1. При изображении графов будем следовать стандартному соглашению в литературе по слабым моделям памяти и обозначаем функцию rf в виде стрелки, идущей от записи к чтению. Аналогично изображено po, а t и i опускаются. Переменные разделяемые между потоками обозначаются с помощью букв x, y и т.д.. Локальные переменные для каждого потока обозначаются буквами a, b и т.д..

Вершинам в графе соответствуют эффекты производимые программой на память разделяемую между потоками. Например, записи $x := 1$ будет соответствовать событие с меткой $\mathtt{W}(x, 1)$, а чтению $a := x$ — $\mathtt{R}(x)$. Если в графе выполняется $G.\mathtt{rf}(r) = w$, то это значит, что при исполнении программы чтение r прочитало свое значение из записи w. Если выполнено $G.\mathtt{po}(e_1, e_2)$ то можно считать что e_1 и e_2 находятся в одном потоке, и e_1 произошло раньше чем e_2. $G.\mathtt{co}$ в каком-то смысле упорядочивает во времени все записи в одну и ту же локацию.

Например, первый граф кодирует множество исполнении, в котором оба чтения (3) и (4) из x и y прочитали инициализирующее значение. А последний граф, кодирует множество исполнений, в котором чтение (3) прочитало значение из записи (1), а чтение (4) из (2)

1.2. Модели памяти

Запуская одну и ту же программу на разных языках или на разных процессорах, можно наблюдать разные поведения. Так происходит, потому что

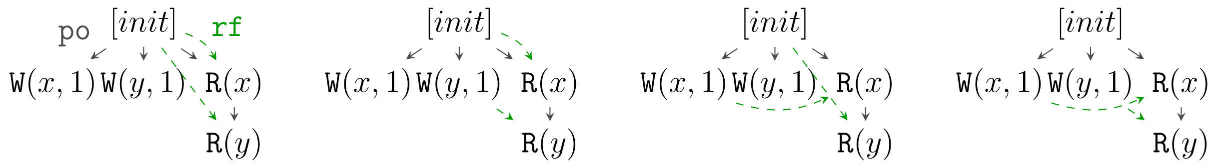

Рис. 1: Графы соответствующие программе 1

компилятор, а потом и сам процессор, могут проводить разного рода оптимизации. Эти оптимизации изначально были написаны для однопоточного кода, поэтому они могут привести к тому, что программа поведет себя совсем не так как ожидалось в случае многопоточного кода. Отличия в поведениях программ как раз обуславливаются тем, что при разных окружениях эти оптимизации разные. Модели поведения многопоточных программ называются моделями памяти,далее ММ (memory model). Изучая конкретную ММ, можно сформулировать некоторый критерий, с помощью которого по графу исполнения программы можно будет говорить допустим он или нет в рамках данной ММ. Поэтому ММ часто моделируются просто предикатами на графах.

Пусть дана некоторая модель памяти m. Соответствующий ей предикат будем обозначать $\texttt{cons}_m$. Будем говорить, что граф G консистентен относительно m, если $\texttt{cons}_m(G)$ верно.

1.3. Алгоритм TruSt

Алгоритм TruSt представлен на рис. 1.3. Получив на вход программу P, Verify начинает исследовать P, вызывая Visit с графом исполнения G_0, содержащим только события инициализации. Далее Visit начинает поиск в глубину всех графов исполнения P. При этом поиске для каждого графа Visit убеждается, что он не содержит события с меткой `Error`, обозначающей ошибку.

Давайте теперь более подробно рассмотрим функцию Visit, которая является ядром TruSt. На каждом шаге, если текущий граф выполнения G консистентен относительно данной модели памяти (строка 4), Visit расширяет текущий граф G, вызывая $\mathsf{next}_P(G)$.

Функция $\mathsf{next}_P(G)$ находит поток, в котором еще можно исполнить хотя бы одну инструкцию, исполняет ее и добавляет соответствующее ей событие a

1: **procedure** Verify(P)
2:     Visit(P, G_0)

3: **procedure** Visit(P, G)
4:     **if** $\neg\texttt{cons}_m(G)$ **then return**
5:     **switch** $a \leftarrow \mathsf{next}_P(G)$ **do**
6:         **case** $a = \bot$
7:             **return**
8:         **case** $a \in \texttt{Error}$
9:             **return** “error”
10:         **case** $a \in \texttt{R}$
11:             **for** $w \in G.\texttt{W}_a$ **do**
12:                 Visit(P, $\mathsf{SetRF}(G, a, w)$)
13:         **case** $a \in \texttt{W}$
14:             VisitCOs(P, G, a)
15:             **for** $r \in G.\texttt{R}_a$ such that $(r, a) \notin G.\texttt{porf}$ **do**
16:                 $Deleted \leftarrow \{e \in G.\texttt{E} \mid r <_G e \wedge (e, a) \notin G.\texttt{porf}\}$
17:                 **if** $\forall e \in Deleted \cup \{r\}.$ IsMaximal(G, e, a) **then**
18:                     VisitCOs(P, $\mathsf{SetRF}(G|_{G.\texttt{E} \setminus Deleted}, r, a), a$)

19: **procedure** VisitCOs(P, G, a)
20:     **for** $w_p \in G.\texttt{W}_a$ **do** Visit(P, $\mathsf{SetCO}(G, w_p, a)$)

21: **procedure** IsMaximal(G, e, w)
22:     $B \leftarrow \{w' \in G.\texttt{E} \mid w' \leq_G e \vee (w', w) \in G.\texttt{porf}\}$
23:     $e' \leftarrow$ **if** $e \in G.\texttt{R}$ **then** $G.\texttt{rf}(e)$ **else** e
24:     **return** $e' \in B \wedge \nexists w' \in B.\ (e', w') \in G.\texttt{co}$

(такое событие называется *доступным*) в $G.\texttt{E}$, но не в $G.\texttt{rf}$ или $G.\texttt{co}$ (строка 5). Технически предполагается, что существует некоторый общий порядок $<_{next}$ на событиях, руководствуясь которым $\mathsf{next}_P(G)$ выбирает одно событие из всех доступных (далее будем считать, что $\mathsf{next}_P(G)$ выбирает наименьшее доступное событие относительно порядка $<_{next}$). Кроме того, мы предполагаем, что $<_{next}$ не противоречит порядку выполнения программы (т. е. po $\subseteq <_{next}$). Если доступных событий нет, будем говорить, что граф исполнения G *полон*, и $\mathsf{next}_P(G)$ возвращает $\bot$.

Далее алгоритм разбирает случаи по a.

- Если a равно $\bot$ или `Error`, Visit выходит из цикла (строка 6) или

возвращает ошибку (строка 8) соответственно.

- Если a является чтением, Visit должен посчитать все возможные варианты $\texttt{rf}$ для добавленного события. С этой целью для каждой записи w с такой же локацией как и в a (строка 11), он рекурсивно вызывается на графе, который получается, если расширить $G.\texttt{rf}$, парой (r, w). Любой выбор, ведущий к неконсистентному графу, будет впоследствии устранен проверкой консистентности в строке 4 соответствующего рекурсивного вызова.

- Самый сложный случай возникает, если a является записью. Ясно, что надо обработать случаи, когда уже добавленные в граф чтения r с такой же локацией как и в a, читают из новой записи. Однако, проблема в том, что, то какие события могут быть добавлены после чтения r, зависит от того, какое значение оно будет читать. Поэтому прежде чем, обрабатывать такие случаи, надо удалить все события, добавленные после r.

Чтобы лучше разобраться с последним случаем, рассмотрим пример.

$$(1)x := 1; \;\left\|\; \begin{array}{l} (2)a := x; \\ (3)\texttt{if}\ a\ \texttt{then}\ x := 30 \end{array} \;\right\|\; (4)x := 0$$

Если сначала добавить события (1) и (2), то `if` выполнится и на момент добавления события (4), в графе будет событие $\texttt{W}(x, 30)$. Далее при добавлении события (4), чтобы корректно обработать случай чтения (2) из (4), нам придется удалить (3) из графа. В таких случаях будем говорить, что алгоритму надо сделать *перепосещение*.

Теперь вернемся к алгоритму Visit. Сначала обрабатывается исполнение, в котором все чтения остаются без изменений (строка 14). В этом случае, для каждого возможного $\texttt{co}$-предшественника a — w_p, VisitCOs вставляет a сразу после w_p в порядок $G.\texttt{co}$ с помощью функции $\mathsf{SetCO}(G, w_p, a)$. Формально $\mathsf{SetCO}(G, w_p, w)$ возвращает граф G', который отличается от графа G только

co компонентой:

$$G'.\texttt{co} = G.\cup\{(w', w) \,|\, (w', w_p) \in G.\texttt{co}\} \cup \{(w_p, w)\} \cup \{(w, w') \,|\, (w_p, w') \in G.\texttt{co}\}$$

Теперь посмотрим как происходит перепосещение. Кандидатами для чтений, которых нам надо перепосетить, будут все чтения с той же локацией что и в a, несвязанные с a отношением porf. (Чтения, которые являются porf предками a, исключаются, потому что их перепосещение создало бы porf−цикл, что запрещено консистентностью графа). Для каждого кандидата на чтение r функция Visit вычисляет набор событий, которые будут удалены из G при перепосещении r (строка 16), и проверяет, было ли r и каждое другое событие e, подлежащее удалению, co-максимальным при добавлении, вызвав IsMaximal (строка 17). Если да, то Visit удаляет соответствующие события из G, заставляет r читать из a, а затем вызывает VisitCOs для исследования всех возможных новых порядков co (строка 18).

Теперь посмотрим какая интуиция стоит за процедурой IsMaximal. Рассмотрим консистентный граф G', который может возникнуть в результате перепосещения события чтения r при добавлении события записи w. Если данная модель памяти префикс-замкнута (см. определение 6) знаем, что $G'' := G' \backslash \{r, w\}$ консистентен. Начиная с G'' и предполагая фиксированный порядок $<_{next}$, если добавить все остальные события, можно будет в общем случае прийти к множеству графов $G_1, G_2, ..., G_n$. Дело в том, что в зависимости от того, как эти события были добавлены (например, как были расставлены rf-ребра и т. д.) получаются разные графы. В принципе, все эти графы, могут привести к перепосещению r из w. Наша цель состоит в том, чтобы разрешить такое перепосещение только в одном из графов, а также гарантировать, что такой граф существует. Это достигается, путем требования, чтобы (1) все события в G добавлялись co-максимальным образом и (2) не происходило повторного перепосещения при построении G. Единственность вытекает из того, что существует только одно rf/co-расширение, которое сделало бы новое событие co-максимальным, если не происходит повторного посещения. В то время как существование получается, если наша модель памяти максимально расширяема (см. определение 7): поскольку G'' консистентен, его можно

расширить $\texttt{co}$-максимальным образом.

Формально, говорится, что событие записи $w \in G.\texttt{W}$ $\texttt{co}$-максимально относительно множества событий E, если $w \in E$, и не существует $w' \in E$ такого, что $(w, w') \in G.\texttt{co}$. Событие чтения $r \in G.\texttt{R}$ $\texttt{co}$-максимально относительно E, если $G.\texttt{rf}(r)$ $\texttt{co}$-максимально относительно E. Событие $e \in G.\texttt{E}$ называется максимально-добавляемым перед событием записи $w \in G.\texttt{W}$, если e $\texttt{co}$-максимально относительно множества $Previous_G(e, w) := \{e' \in G.\texttt{E} \mid e' \leqslant_G e \vee (e', w) \in G.\texttt{porf}\}$.

Определение 4 (Максимальное расширение)**.** *Граф исполнения G является максимальным расширением потенциального перепосещения $r \in G.\texttt{R}$ при добавлении $w \in G.\texttt{W}$, если каждое $e \in G.\texttt{E}$, такое что $r \leqslant_G e$ и $(e, w) \in G.\texttt{porf}$, максимально-добавлено перед w.*

IsMaximal как раз и совершает проверку на максимальное расширение.

1.4. Границы применимости TruSt

Введем несколько определений на моделях памяти.

Определение 5 (Правильная сформированность)**.** *Будем говорить что m правильно сформирована если выполняются два условия*

- *для всех $G \sim G'$, $\texttt{cons}_m(G) \Leftrightarrow \texttt{cons}_m(G')$, то есть консистентность графа не зависит от того, каким образом занумерованы его вершины*
- *для всех G, $\texttt{cons}_m(G)$ влечет ацикличность $G.\texttt{porf}$, то есть в графе не может быть циклических зависимостей по данным*

Определение 6 (Префикс-замкнутость)**.** *Будем говорить что m префикс-замкнуто, если для любого G консистентного относительно m, любое его сужение на $G.\texttt{porf}$-замкнутое подмножество тоже консистентно относительно m. Иными словами, любой подграф консистентного графа консистентен*

Определение 7 (Максимальная расширяемость)**.** *Будем говорить что m максимально расширяемо, если при расширении любого консистентного графа* `po`- *и* `co`-*максимальным событием получается тоже консистентный граф. Иными словами, добавление новых событий "в конец"графа, сохраняет его консистентность.*

Все дальнейшие теоремы будут доказаны в предположении, что данные модели памяти удовлетворяют определениям 5, 6 и 7. Кроме того будем считать что все данные нам программы конечны.

Определение 8 (Конечность программы)**.** *Будем говорить, что программа P конечна, если существует такая константа K, что для любого графа G, соответствующего какому-то исполнению P верно*

$$|G.\mathtt{E}| < K$$

Ясно, что все программы не имеющие циклов и таких операторов как `goto` являются конечным. Чтобы частично верифицировать многопоточную программу с циклом, можно развернуть этот цикл на некоторую глубину k и провести верификацию по модулю этой глубины.

1.5. Язык Coq

Coq — средство автоматической проверки доказательств. Это чистый функциональный язык программирования с мощной системой зависимых типов.

Несмотря на то, что существует большое количество систем интерактивной проверки доказательств (например, Isabelle/HOL [17], Lean [18], Agda [19], Arend [20]...) для этого проекта был выбран именно Coq, из-за его богатой экосистемы и большого сообщества.

С помощью зависимых типов в Coq можно закодировать формулировки теорем. Далее, на термы этого языка можно смотреть как на доказательства утверждений. Поэтому задача доказательства теорем в Coq сводится к предъявлению терма, населяющего данный тип. Проверка корректности доказательств в этом случае полностью перекладывается на алгоритм проверки типов.

Так как в основе ядра Coq лежит соответствие Карри-Говарда, термы языка выступают не только в качестве доказательств, но и в качестве функциональных программ. Это позволяет кодировать на нем различные алгоритмы, в том числе и TruSt. Далее про эти алгоритмы можно формулировать и доказывать теоремы.

Термы в этом языке можно пытаться предъявить явно, но удобнее всего это делать с помощью тактик. Тактики — это специальные макросы, которые позволяют генерировать различные термы. Хоть в общем случае задача поиска доказательств неразрешима, тактики позволяют автоматически ее решать в некоторых случаях, например для разрешимых арифметических теорий [21].

Так как проверку доказательств осуществляет тайпчекер, то корректность механизированных доказательств полностью зависит от его корректности. Поэтому очень важно, чтобы у интерактивного средства доказательств тайпчекер был компактным и хорошо задокументированным. У языка Coq тайпчекер состоит из примерно 20 тыс. строк на языке OCaml [23], что считается довольно компактным ядром.

Хотя за корректность процедуры проверки доказательств в Coq отвечает компактное ядро, на пользователе лежит еще одна очень важная задача. А именно, задача правильной спецификации утверждений, которые нужно доказать. Утверждения закодированные внутри системы доказательства теорем должны быть понятны и сильно похожи на их неформальные аналоги. Только тогда механизированные доказательства могут быть приняты сообществом.

1.6. Библиотека Hahn

На Coq реализованно очень много проектов, связанных с верификацией различных алгоритмов, поэтому его экосистема подходит для наших целей. В частности в работе была использована специальная библиотека Hahn [22] для Coq, разработанная командой института Макса Планка, под руководством Виктора Вафеядиса. Эта библиотека была задействована для механизации результатов в области слабых моделей памяти, например в реализациях модели памяти IMM [25], денотационной семантики конкурентных программ [26] и т.д..

В Hahn находятся определения специфичные для работы с семантикой конкурентных программ, например определения графов исполнения и основных слабых моделей памяти, таких как SC, TSO, PSO. Кроме того в ней содержится большое количество инструментов для работы с отношениями. Используя эту библиотеку получается формулировать теоремы в Coq очень близким образом к тому как они выглядят в математической нотации. Кроме того, библиотека предоставляет большое количество тактик для автоматизации рутинной работы.

2. Теоремы о Корректности и Терминируемости

В этой и последующих главах приведены основные схемы доказательств теорем о правильности работы алгоритма TruSt. Полные формализованные доказательства теорем выполненные в системе Coq находятся в репозитории: https://github.com/volodeyka/trust-coq.

В данной главе приведены схемы доказательств теорем о корректности и терминируемости алгоритма TruSt. В ходе доказательства последней теоремы был использован подход поиска некоторой ограниченной сверху величины, которая строго увеличивается с каждым рекурсивным вызовом алгоритма. Далее, так как количество рекурсивных вызовов алгоритма на каждом этапе конечно, получаем, что алгоритм в самом деле завершается. Чтобы реализовать данный подход алгоритм был закодирован как отношение. Далее было показано что любая возрастающая цепочка в получившемся отношении обрывается.

2.1. TruSt в терминах отношений

Для начала покажем как кодировать алгоритм TruSt в виде отношения.

Напомним, что любой граф исполнения G, индуцирует некоторый тотальный порядок $<_G$ на $G.\mathtt{E}$, так как $G.\mathtt{E}$ является последовательностью вершин. Далее будем опускать индекс G у $<_G$ и писать просто $<$, когда из контекста понятно о каком графе идет речь. Более того будем писать $<'$ вместо $<_{G'}$, $<_1$ вместо $<_{G_1}$ и так далее...

Определение 9 (Отношение $\Rightarrow$)**.** *Пусть у нас фиксированна некоторая программа P, будем писать $G \Rightarrow G'$ и говорить что алгоритм делает шаг G из G', если Visit(P, G) вызывает Visit(P, G') на верхнем уровне рекурсии и G' проходит проверку на консистентность $\mathtt{cons}_m(G)$ на строке 4.*

Кроме того, аннотируем данное отношение так, чтобы можно было отслеживать какое событие было добавлено и понимать произошло ли перепосещение.

Определение 10 (Отношения $\overset{e}{\underset{nr}{\Rightarrow}}$, $\overset{e}{\underset{rv\ a}{\Rightarrow}}$ и $\overset{e}{\underset{t}{\Rightarrow}}$)**.** *Пусть $G \Rightarrow G'$, то есть алгоритм делает шаг из G в G', тогда будем писать*

- $G \overset{e}{\underset{nr}{\Rightarrow}} G'$ *если при этом шаге не произошло перепосещения и мы просто добавили событие* e *(строки 10 и 14), где* nr *расшифровывается как "non-revisit"*

- $G \overset{e}{\underset{rv\ a}{\Rightarrow}} G'$ *если при этом шаге мы хотели добавить событие* e*, и произошло перепосещение чтения* a *(строка 18), где* rv *расшифровывается как "revisit"*

- $G \overset{e}{\underset{t}{\Rightarrow}} G'$ *если* $G \overset{e}{\underset{nr}{\Rightarrow}} G'$ *или для некоторого* a *выполняется* $G \overset{e}{\underset{rv\ a}{\Rightarrow}} G'$*, то есть* $t = nr$ *или* $t = rv\ a$

Для удобства так же будем отслеживать множество записей RW, которые вызвали перепосещение.

Определение 11. *Пусть* $G \overset{e}{\underset{t}{\Rightarrow}} G'$*, будем писать*

- $(RW, G) \overset{e}{\underset{rv\ a}{\Rightarrow}} (RW \cup \{e\}, G')$*, если* $G \overset{e}{\underset{rv\ a}{\Rightarrow}} G'$

- $(RW, G) \overset{e}{\underset{nr}{\Rightarrow}} (RW, G')$*, если* $G \overset{e}{\underset{nr}{\Rightarrow}} G'$

2.2. Теорема о Корректности

Напомним, что теорема о Корректности формулируется так: все сценарии исполнения программы, посещаемые в ходе работы алгоритма, корректны относительно заранее выбранной модели памяти. Теперь переформулируем ее более формально.

Теорема 1 (Теорема о Корректности)**.** *Пусть для некоторого графа* G *выполняется* $\emptyset \Rightarrow^* G$*, тогда верно* $\texttt{cons}_G(m)$

Доказательство. Эта теорема доказывается по определению отношения $\Rightarrow$, так как Visit явно проверяет консистентность на каждом шаге. □

2.3. Теорема о Терминируемости

Сначала дадим высокоуровневое объяснение завершаемости TruSt. Завершение алгоритма TruSt следует из предположения, что все графы исполнения программы P имеют ограниченный размер. Предположим, что алгоритм

зависает, значит в нем есть бесконечная цепочка вложенных вызовов рекурсии, то есть бесконечная возрастающая цепочка $\emptyset \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \ldots$ Поскольку все шаги алгоритма, за исключением перепосещений, увеличивают размер графа, а он ограничен, заключаем, что среди этих шагов есть бесконечное множество перепосещений. Но, как будет доказано, события, добавленные при перепосещении, не могут быть удалены, значит размер графа строго растет, а это противоречит его ограниченности.

Теперь приведем формальную схему доказательства данного утверждения.

Лемма 1. *Если* $(\emptyset, \emptyset) \Rightarrow^* (RW, G)$*, то* $RW \subseteq G.\mathtt{E}$

Доказательство. Сначала покажем, что после добавления любой записи w, вызывающей перепосещение некоторого чтения r (все такие записи как раз складываются в множество RW из определения 2.1), она не может быть удалена при дальнейшем исполнении алгоритма (заметим, что в этом случае r читает из w).

Предположим противное: что w было удалено. События удаляются только при каком-то перепосещении. Поэтому будем считать что w было удалено при перепосещении чтения r', когда добавлялась некоторая запись w'. Разберем два случая

- Если $r' = r$ или было добавлено до r, то разобрав различные случаи в соответствии с определением $Deleted$ (строка 16), можно показать, что r должно быть в наборе удаляемых событий при перепосещении r', иначе w не будет удалено. Но само r не может быть удалено в таком случае, потому что оно не было максимально добавлено перед w' (оно читает из события w, которое было добавлено до w').

- В случае если r' добавлено после r, по определению $Deleted$ на строке 16, не будет удалено r, а значит и не будет удалено w, потому что из него читает неудаляемое событие (r).

Теперь наше утверждение легко доказать по индукции:

- в начале $RW = \emptyset \subseteq \emptyset = G.\mathtt{E}$

- теперь предположим, что $(\emptyset,\emptyset) \Rightarrow^* (RW, G) \overset{e}{\underset{t}{\Rightarrow}} (RW', G')$ и $RW \subseteq G.\texttt{E}$. Разберем два случая:

  – $t = nr$, тогда, очевидно, что все выполняется так как

$$RW' = RW \subseteq G.\texttt{E} \subseteq G.\texttt{E} \cup \{e\} = G'.\texttt{E}$$

  – $t = rv\ a$, тогда как известно ни одно событие из RW не было удалено и

$$RW' = RW \cup \{e\} \subseteq G'.\texttt{E}$$

□

Известно, что на множестве $\mathbb{N} \times \mathbb{N}$ есть строгий лексикографический порядок:

$$(a, b) < (c, d) :\Leftrightarrow a < c \vee (a = c \wedge b < d)$$

Тогда не сложно доказать, что величина $(|RW|, |G.\texttt{E}|)$ будет строго увеличиваться (по отношению к этому порядку) с каждым шагом алгоритма.

Лемма 2. *Пусть* $(\emptyset,\emptyset) \Rightarrow^* (RW, G) \overset{e}{\underset{t}{\Rightarrow}} (RW', G')$*, тогда*

$$(|RW|, |G.\texttt{E}|) < (|RW'|, |G'.\texttt{E}|)$$

Доказательство. Рассмотрим случаи

- $t = nr$, тогда $|RW| = |RW'|$, и $|G'.\texttt{E}| = |G.\texttt{E}| + 1 > |G.\texttt{E}|$, поэтому выполняется утверждение леммы
- $t = rv\ a$, тогда $|RW'| = |RW| + 1 > |RW|$

□

Теперь заметим, что алгоритм посещает граф исполнения G тогда и только тогда, когда $\emptyset \Rightarrow^* G$, а значит определение 1.4 можно переформулировать следующим образом.

Определение 12 (Конечность программы)**.** *Существует* $K \in \mathbb{N}$*, такое что,* $\emptyset \Rightarrow^* G$*, влечет*

$$|G.\mathtt{E}| < K$$

Тогда в предположении этого свойства нетрудно заключить следующий факт.

Лемма 3. *Величина из предыдущей леммы ограничена парой* (K, K)

Доказательство. По лемме 2.3, $RW \subseteq G.\mathtt{E}$, а значит $|RW| \leqslant |G.\mathtt{E}| < K$ □

Теорема 2 (Терминируемость TruSt)**.** *Алгоритм TruSt завершается*

Доказательство. Предположим, что алгоритм зависает, значит в нем есть бесконечная цепочка вложенных вызовов рекурсии, то есть бесконечная возрастающая цепочка

$$\emptyset \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \ \dots$$

Но можно доказать, что существует такая константа $B \in \mathbb{N}$, что для всех графов исполнения G, $\emptyset \Rightarrow^k G$ влечет $k < B$.

Это верно так как $\emptyset \Rightarrow^k G$, влечет то, что существует RW, такое что $(\emptyset, \emptyset) \Rightarrow^k (RW, G)$, а значит по предыдущим двум леммам $k < K^2$ □

3. Теоремы о Полноте и Оптимальности

В данной главе приведены схемы доказательств теоремы о полноте (то есть, о том, что алгоритм посещает все консистентные графы) и оптимальности (то есть, о том, что алгоритм не посещает один граф более одного раза) TruSt. Для этого был придуман алгоритм Prev, являющийся обратным к TruSt. Далее Prev был закодирован с помощью детерминированного отношения. В итоге было показано, что из того, что Prev обратен к TruSt справа следует полнота, а из обратности слева следует оптимальность.

3.1. Алгоритм Prev и его базовые свойства

В этой главе зафиксируем некоторую программу P и модель памяти m, удовлетворяющие вышеперечисленным условиям.

Для начала введем понятие максимального события в графе исполнения.

Определение 13 (Максимальное событие)**.** *Пусть дан граф G, определим максимальное событие $me(G)$, как максимум по отношению к $<_{next}$, среди всех* $\texttt{porf}$ *максимальных событий*

$$me(G) := \max_{<_{next}} \{e \in G.\texttt{E} \mid codom([e]; G.\texttt{porf}) = \emptyset\}$$

Для $me(G)$ верна следующая лемма

Лемма 4. *Для любого конечного графа G*

$$me(G) = \emptyset \Leftrightarrow G = \emptyset$$

Также нам пригодится определение полного графа.

Определение 14 (Полный граф)**.** *Будем говорить что граф G_f является полным и писать $full(G_f)$, если выполняется следующий набор условий*

- $\texttt{cons}_m(G_f)$
- *G_f соответствует некоторому исполнению программы P*

- $\mathsf{next}_P(G) = \bot$

Теперь, чтобы лучше понять алгоритм Prev сформулируем следующую лемму.

Лемма 5 (Максимальное событие после шага TruSt)**.** *Пусть даны графы G и G', такие что $\emptyset \Rightarrow^* G \overset{e}{\underset{t}{\Rightarrow}} G'$. Тогда если*

- *$t = nr$, верно что $me(G') = \{e\}$*
- *$t = rv\ a$, для некоторого a, верно что $me(G') = \{a\}$*

Доказательство. Доказательство получается разбором случаев в шаге TruSt из G в G', и было опущено. Его можно найти в Coq-репозитории. □

Теперь рассмотрим алгоритм Prev, представленный ниже. Сначала берем максимальное событие a (строка 2). Далее нужно понять было ли оно чтением из-за которого произошло перепосещение или добавлено при обычном шаге алгоритма. Для этого, если a было чтением, проверяем, что запись e, из которой читает a, $<_{next}$ меньше a, и в графе нет событий porf после e, кроме a (строка 6). Если данная проверка была пройдена, то очевидно, что граф G, мог получиться при перепосещении из некоторого графа G'. Так как co-максимальность гарантирует существование и единственность такого G', его можно построить с помощью процедуры MaxCompletion.

Если проверка не пройдена или же максимальное событие не является чтением, то граф G, по лемме 5, не мог быть получен при перепосещении и Prev возвращает граф $G \setminus \{a\}$.

Теперь посмотрим на процедуру MaxCompletion. Так как при перепосещении в TruSt делается проверка каждого удаляемого события на co-максимальность, восстановить граф G' будет несложно: надо просто добавлять события co-максимальным образом пока не встретится e. Иными словами, берем следующее событие a (строка 9), и если это запись добавляем ее так, чтоб она была co-максимальной (строка 15), а если это чтение, добавляем его так, чтобы оно читало из максимальной записи (строка 13).

Теперь закодируем Prev как отношение.

```
1:  procedure Prev(G)
2:      a ← me(G)
3:      if a ∈ R then
4:          e ← G.rf(a)
5:          if a <_next e ∧ codom([e]; G.porf) = {a} then
6:              return (MaxCompletion(G|_{G.E\{a,e}}, e), e, rv a)
7:      return (G \ {a}, a, nr)
8:  procedure MaxCompletion(G, e)
9:      a ← next(G)
10:     while a ≠ e do
11:         switch a do
12:             case a ∈ R
13:                 G ← SetRF(G + a, a, GetMaximalWrite(G, a))
14:             case a ∈ W
15:                 G ← SetWriteMaximal(G + w, w)
16:             case _
17:                 G ← G + a
18:     return G
19: procedure GetMaximalWrite(G, a)
20:     return max_{G.co} {G.W_loc(a)}
21: procedure SetWriteMaximal(G, w)
22:     return SetCO(G, GetMaximalWrite(G, w), w)
```

Определение 15. *Будем говорить что Prev делает шаг из G в G', и писать $G \underset{t}{\overset{e}{\leadsto}} G'$, если*

$$Prev(G') = (G, e, t)$$

Далее покажем два свойства, которые понадобятся для доказательств оптимальности и полноты

Лемма 6 (Детерменированность $\leadsto$). *Пусть дан некий граф G и некоторые G_1, G_2, такие что $G_1 \underset{t_1}{\overset{e_1}{\leadsto}} G$ и $G_2 \underset{t_2}{\overset{e_2}{\leadsto}} G$, тогда*

$$G_1 = G_2,\ e_1 = e_2\ \text{ и }\ t_1 = t_2$$

Доказательство. Этот факт следует из определения $\leadsto$ □

Теорема 3 (Терминируемость $\rightsquigarrow$)**.** *Пусть дан некий граф G, для которого есть полный граф G_f, такой что $G \rightsquigarrow^* G_f$. Тогда*

$$\emptyset \rightsquigarrow^* G$$

Доказательство этой теоремы более сложное и требует вспомогательных лемм.

Для начала, можно показать следующий факт.

Лемма 7. *Пусть дан некий непустой граф G, такой что $G \rightsquigarrow^* G_f$, для полного графа G_f, тогда существует G', такой что*

$$G' \rightsquigarrow G$$

Доказательство. Можно показать, что для таких графов как G цикл while при вызове MaxCompletion не зависает (доказательство этого факта можно найти в Coq-репозитории), и тогда, пользуясь леммой 4, выводится условие нашей леммы. □

Теперь по аналогии с TruSt можно найти меру, строго убывающую с каждым шагом алгоритма. Для этого проаннотируем отношение $\rightsquigarrow$ множеством записей, при добавлении которых произошло перепосещение.

Определение 16. *Пишем*

- $(RW \cup \{e\}, G) \underset{rv\ a}{\overset{e}{\rightsquigarrow}} (RW, G')$, *если* $G \underset{rv\ a}{\overset{e}{\rightsquigarrow}} G'$, *и*
- $(RW, G) \underset{nr}{\overset{e}{\rightsquigarrow}} (RW, G')$, *если* $G \underset{nr}{\overset{e}{\rightsquigarrow}} G'$

После чего, можно показать, что выполняется следующая лемма.

Лемма 8. *Если* $(RW, G) \underset{rv\ a}{\overset{e}{\rightsquigarrow}} (RW', G')$, *то* $|RW| > |RW'|$

И далее по аналогии с TruSt можем легко получить терминируемость алгоритма Prev и доказать теорему 3.

3.2. Теорема о Полноте

Главная идея в доказательстве теоремы о полноте TruSt заключается в том, чтобы доказать, что Prev обратен к TruSt справа (точная формулировка приведена в следующей лемме) и воспользоваться теоремой 3.

Ключевую роль тут играет следующая лемма.

Лемма 9. *Пусть дан полный граф G_f и некоторые графы G и G'. Так же пусть выполняется*

- $\emptyset \Rightarrow^* G$
- $G \overset{e}{\underset{t}{\rightsquigarrow}} G'$
- $G' \rightsquigarrow^* G_f$

тогда

$$G \overset{e}{\underset{t}{\Rightarrow}} G'$$

Доказательство. Доказательство получается разбором случаев, и было опущено. Его можно найти в Coq-репозитории. □

Теперь, собрав все факты воедино можем доказать теорему о полноте TruSt.

Теорема 4 (Полнота TruSt)**.** *Для любого графа G_f, из $full(G_f)$, следует*

$$\emptyset \Rightarrow^* G_f$$

Доказательство. Раз граф G_f полон, то по теореме 3 получаем, $\emptyset \rightsquigarrow^* G_f$. Далее можно доказать индукцией по k, что из $\emptyset \rightsquigarrow^k G \rightsquigarrow^* G_f$ следует $\emptyset \Rightarrow^* G$:

- если $k = 0$, то нам надо доказать $\emptyset \Rightarrow^* \emptyset$, что очевидно
- далее пусть известно, что $\emptyset \rightsquigarrow^k G \rightsquigarrow G' \rightsquigarrow^* G_f$ и $\emptyset \Rightarrow^* G$. По предыдущей лемме $G \Rightarrow G'$, из чего легко следует индукционный переход.

Из последнего факта следует теорема о полноте. □

3.3. Теорема об Оптимальности

Главная идея в доказательстве теоремы об оптимальности TruSt заключается в том, чтобы доказать, то что Prev обратен к TruSt слева (точная формулировка приведена в следующей лемме) и воспользоваться теоремой 6.

Лемма 10. *Пусть даны некоторые графы G и G'. Так же пусть выполняется*

- $\emptyset \Rightarrow^* G$
- $G \underset{t}{\overset{e}{\Rightarrow}} G'$

тогда

$$G \underset{t}{\overset{e}{\rightsquigarrow}} G'$$

Доказательство. Доказательство получается разбором случаев, и было опущено. Его можно найти в Coq-репозитории. □

Теорему об оптимальности будем формулировать и доказывать в терминах трасс алгоритма TruSt. Иными словами докажем, что если есть две трассы алгоритма TruSt, заканчивающиеся на один и тот же граф, то сами трассы совпадают.

Теорема 5 (Оптимальность TruSt)**.** *Пусть даны две трассы алгоритма TruSt.*

- $\emptyset \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \ \ldots \Rightarrow G_n$
- $\emptyset \Rightarrow H_1 \Rightarrow H_2 \Rightarrow \ \ldots \Rightarrow H_m$

Кроме того они заканчиваются на одинаковые графы, то есть $G_n = H_m$, тогда

$$n = m, \ \text{и для всех } i, \ G_i = H_i$$

Доказательство. Будем доказывать утверждение индукцией по длине первой трассы:

- в случае когда у первой трассы длинна 0, имеем, что $H_m = \emptyset$, и нам достаточно доказать, что $m = 0$. Пусть это не так и для некоторого k, $m = k + 1$, тогда

$$\emptyset \Rightarrow H_1 \Rightarrow H_2 \Rightarrow \ \ldots \Rightarrow H_k \Rightarrow \emptyset$$

Но это означает, что $\emptyset$ является петлей для отношения $\Rightarrow^{k+1}$. Тогда для любого N, $\emptyset \Rightarrow^{(k+1)N} \emptyset$, а это противоречит теореме о терминируемости TruSt.

- в случае, когда $n = k + 1$, для некоторого k, нам надо рассмотреть два варианта

  – $m = 0$. Тогда теорема доказывается аналогично тому, как была доказана база индукции.

  – $m = l + 1$, для некоторого l. Тогда пользуясь леммой 10 понимаем, что существуют e, t, e' и t', такие что

  $$G_k \underset{t}{\overset{e}{\leadsto}} G_{k+1} \text{ и } H_l \underset{t'}{\overset{e'}{\leadsto}} H_{l+1}$$

  Далее, учитывая $G_{k+1} = H_{l+1}$, по лемме 6 получаем, что $G_k = H_l$ и остается просто воспользоваться индукционным предположением.

□

Заключение

В рамках данной дипломной работы были достигнуты следующие результаты.

- Было доказано, что алгоритм TruSt корректен, то есть он в ходе работы посещает только те графы исполнения, которые соответствуют заранее выбранной модели памяти.
- Была доказана терминируемость алгоритма TruSt на конечных программах, то есть он завершается если на вход дана программа с ограниченными графами исполнения
- Была доказана полнота алгоритма TruSt в случае конечных программ, то есть он посещает все графы исполнения конечных программ, соответствующие заранее выбранной модели памяти
- Была доказана оптимальность алгоритма TruSt в случае конечных программ, то есть он не посещает они и те же граф исполнения более одного раза

Все теоремы были механизированы в системе Coq.

Список литературы

[1] Model checking for programming languages using VeriSoft. In: POPL 1997. Paris, France: ACM, pp. 174-186. doi: https://doi.org/10.1145/263699.263717.

[2] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu (2008). Finding and reproducing Heisenbugs in concurrent programs. In: OSDI 2008. USENIX Association, pp. 267-280.

[3] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas (2014). Optimal dynamic partial order reduction. In: POPL 2014. New York, NY, USA: ACM, pp. 373-384.

[4] Cormac Flanagan and Patrice Godefroid (2005). Dynamic partial-order reduction for model checking software. In: POPL 2005. New York, NY, USA: ACM, pp. 110-121.

[5] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis (2019). Model checking for weakly consistent libraries. In: PLDI 2019. New York, NY, USA: ACM.

[6] Antoni Mazurkiewicz (1987). Trace Theory. In: PNAROMC 1987. Vol. 255. LNCS. Berlin, Heidelberg: Springer, pp. 279-324

[7] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas (2015). Stateless model checking for TSO and PSO. In: TACAS 2015. Vol. 9035. LNCS. Berlin, Heidelberg: Springer, pp. 353-367.

[8] Elvira Albert, Puri Arenas, Maria Garcia de la Banda, Miguel Gomez-Zamalloa, and Peter J. Stuckey (2017). Conmath-sensitive dynamic partial order reduction. In: CAV 2017. Ed. by Rupak Majumdar and Viktor Kuncak. Cham: Springer International Publishing, pp. 526-543

[9] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya (Dec. 2017). Data-centric dynamic partial order reduction. In: Proc. ACM Program. Lang. 2.POPL, 31:1-31:30.

[10] Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman (Oct. 2019). Value-Centric Dynamic Partial Order Reduction. In: Proc. ACM Program. Lang. 3.OOPSLA.

[11] Naling Zhang, Markus Kusano, and Chao Wang (2015). Dynamic partial order reduction for relaxed memory models. In: PLDI 2015. New York, NY, USA: ACM, pp. 250-259.

[12] Leslie Lamport (Sept. 1979). How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. In: IEEE Trans. Computers 28.9, pp. 690-691.

[13] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen (July 2010). X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. In: Commun. ACM 53.7, pp. 89-97.

[14] SPARC International Inc. (1994). The SPARC architecture manual (version 9). Prentice-Hall. Naling Zhang, Markus Kusano, and Chao Wang (2015). Dynamic partial order reduction for relaxed memory models. In: PLDI 2015. New York, NY, USA: ACM, pp. 250-259

[15] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer (2017). Repairing sequential consistency in C/C++11. In: PLDI 2017. Barcelona, Spain: ACM, pp. 618-632.

[16] Сайт проекта Coq. — URL: https://coq.inria.fr.

[17] Сайт проекта isabelle. – URL: https://isabelle.in.tum.de

[18] Сайт проекта Lean. – URL: https://leanprover-community.github.io

[19] Сайт проекта Agda. – URL: https://github.com/agda/agda

[20] Сайт проекта Arend. – URL: https://arend-lang.github.io

[21] Мануал по решателям арифметики в Coq. – URL: https://coq.inria.fr/refman/addendum/micromega.html

[22] Сайт библиотеки Hahn для Coq – URL: https://github.com/vafeiadis/hahn

[23] Сайт языка OCaml – URL: https://ocaml.org

[24] Jain, Kush and Palmskog, Karl and Celik, Ahmet and Gallego Arias, Emilio Jesus and Gligoric, Milos. mCoq: Mutation Analysis for Coq Verification Projects // International Conference on Software Engineering, Tool Demonstrations Track – 2020

[25] Podkopaev, Anton and Lahav, Ori and Vafeiadis, Viktor. Bridging the Gap between Programming Languages and Hardware Weak Memory Models // Proceedings of the ACM on Programming Languages (POPL'19). – 2019. – P. 1-31

[26] Jeffrey, Alan and Riely, James and Batty, Mark and Cooksey, Simon and Kaysin, Ilya and Podkopaev, Anton. The Leaky Semicolon: Compositional Semantic Dependencies for Relaxed-Memory Concurrency // Proceedings of the ACM on Programming Languages (POPL'22). – 2022

[27] Leroy X. Formal verification of a realistic compiler // Communications of the ACM. — 2009. — Vol. 52, no. 7. — P. 107–115.

[28] seL4: Formal verification of an OS kernel / G. Klein, K. Elphinstone, G. Heiser et al. // 22nd Symposium on Operating Systems Principles (SOSP'09). — 2009. — P. 207–220.

[29] A machine-checked proof of the Odd Order Theorem / G. Gonthier, A. Asperti, J. Avigad et al. // 4th International Conference on Interactive Theorem Proving (ITP'13). — 2013.

[30] Using Crash Hoare logic for certifying the FSCQ file system / H. Chen, D. Ziegler, T. Chajed et al. // 25th Symposium on Operating Systems Principles (SOSP'15). — 2015. — P. 18–37.

[31] Gonthier G. A computer-checked proof of the Four Colour Theorem. — 2005.