УДК 004.45
DOI: 10.17587/prin.16.558-569

В. О. Мордань[1], канд. физ.-мат. наук, науч. сотр., mordan@ispras.ru,
В. С. Мутилин[1,2], канд. физ.-мат. наук, науч. сотр., mutilin@ispras.ru,
[1] Институт системного программирования им. В. П. Иванникова Российской академии наук, Москва
[2] Московский физико-технический институт (национальный исследовательский университет)

Исследование значимости элементов трасс ошибок в статической верификации программного обеспечения

Поступила в редакцию 29.05.2025
Принята к публикации 08.07.2025

Рассмотрено одно из ключевых ограничений верификации программного обеспечения — необходимость ручной оценки результатов, включающих предупреждения о потенциальных ошибках. Такие предупреждения должны быть либо исправлены, либо классифицированы как ложные срабатывания, что требует участия экспертов. Для решения описанной проблемы формально введено понятие тщательности трасс ошибок в верификации программного обеспечения. Тщательность трасс оценивается на основе результатов инструментальных средств, участвовавших в соревнованиях по верификации программного обеспечения SV-COMP, что демонстрирует практическое применение предлагаемого подхода к реальным верификационным задачам. Показано, как понятие тщательности может быть использовано в качестве метрики при сравнении инструментальных средств верификации, а также для улучшения автоматического подтверждения результатов верификации. Помимо этого, предложенная метрика показывает, какие именно элементы трассы ошибки понадобятся для ее визуализации.

Ключевые слова: *верификация программного обеспечения, тщательность, результаты верификации, валидация, трасса ошибки*

Для цитирования: Мордань В. О., Мутилин В. С. Исследование значимости элементов трасс ошибок в статической верификации программного обеспечения // Программная инженерия. 2025. Том 16, № 11. С. 558—569. DOI: 10.17587/prin.16.558-569.

Введение

Верификация программного обеспечения является формальным методом автоматической проверки исходного кода программы без ее выполнения, при котором проводится анализ всех возможных путей выполнения. Целью является доказательство корректности программы относительно проверяемых свойств при соответствующих предположениях в отличие от методов тестирования и поиска ошибок, таких как статический анализ. Верификация программного обеспечения поддерживает различные типы свойств, например, недостижимость ошибочных функций или отсутствие гонок по данным. Несмотря на то, что ее основная задача — доказательство корректности, методы верификации также справляются и с обнаружением всех нарушений свойств [1], включая те, которые сложно воспроизвести с помощью тестов. Это делает их особенно подходящими для критически важных программных систем, таких как операционные системы, онлайн-банкинг и авионика.

В то же время практическое применение верификации программного обеспечения сопряжено с рядом сложностей. Во-первых, эта задача в общем случае неразрешима, в результате чего соответствующие подходы и решения могут требовать значительных вычислительных ресурсов и при этом не всегда способны ее решить. На настоящее время инструментальные средства верификации программного обеспечения используют различ-

ные допущения (относительно кода или моделей окружения) для повышения эффективности, однако верификация программной системы все еще может занимать несколько дней [2—5]. Во-вторых, если целью является исправление всех найденных ошибок, то результаты верификации необходимо анализировать вручную, чтобы определить, действительно ли нарушения проверяемого свойства соответствуют реальным ошибкам в программе. Ложные срабатывания могут возникать в силу некорректных спецификаций или неправильного применения допущений в рамках используемого инструментария. Те же аргументы применимы и к доказательствам корректности — пользователи должны вручную их проверять, чтобы убедиться в отсутствии пропущенных ошибок (что является более сложной задачей на практике). Подобный ручной анализ является достаточно трудоемкой процедурой, что значительно увеличивает длительность процесса верификации. Например, при анализе результатов верификации драйверов Linux в рамках инициативы Google Summer of Code были выявлены 62 потенциальные ошибки, однако после трехмесячного детального ручного анализа были подтверждены лишь две ошибки[1] (аналогичные результаты ручного анализа доступны по ссылкам[2,2]). Это подчеркивает критическую важность задачи: пользователям необходимы результаты, содержащие все необходимые элементы для эффективного поиска ошибок, которые будут представлены в удобном для анализа виде. Таким образом, практический успех верификации программного обеспечения в значительной степени зависит от полноты и качества сгенерированных результатов.

Постоянный прогресс в области верификации программного обеспечения приводит к росту как эффективности, так и результативности, что наглядно демонстрируют ежегодные соревнования по верификации программного обеспечения (SV-COMP) [2—4]. Этот прогресс позволяет успешно применять инструментальные средства верификации ко все более широкому спектру программных систем. Например, в соревнованиях SV-COMP'23 участвовали 52 современных средства верификации на бенчмарках, включающих 23 805 программ на языке C и 586 программ на языке Java с известными результатами. Бенчмарки были классифицированы в соответствии с проверяемыми свойствами, и каждый участвующий инструментарий выполнял задачи в условиях ограниченных ресурсов (15 ГБ оперативной памяти, 8 ядер процессора и 15 мин процессорного времени). В качестве результата инструментальные средства выдают файлы в формате GraphML [6], представляющие собой либо доказательства корректности, либо трассы ошибок. Они выполняют двойную функцию: позволяют разработчикам вручную анализировать и исправлять ошибки, а также дают возможность автоматически валидировать результаты с помощью специализированных инструментальных средств, повышая тем самым уверенность в результатах верификации.

Как для автоматической валидации, так и для ручного анализа крайне важно, чтобы трассы содержали ключевые элементы, такие как вызовы функций и оценку значений переменных. Однако некоторые инструментальные средства верификации формируют трассы, лишенные этих важных деталей, что делает валидацию ненадежной, а ручной анализ — практически невозможным. В данной работе авторы ставят следующий требующий исследования вопрос: какие элементы трасс ошибок являются наиболее критичными для процесса валидации? При этом основное внимание уделяется валидации трасс ошибок, тогда как ручной анализ и валидация доказательств корректности остаются за рамками работы и рассматриваются как направления для будущих исследований.

Предлагаемая авторами основная гипотеза заключается в том, что включение ключевых элементов в трассы значительно повышает успешность их валидации. Цель работы — определить эти критически важные элементы, которые станут основой для формализации понятия тщательности трасс ошибок. Это позволит оценивать тщательность трасс, сгенерированных различными инструментальными средствами верификации, и определять, насколько легко они могут быть провалидированы. Измеряя тщательность трасс, авторы стремятся разработать автоматический метод, позволяющий определить, является ли трасса полной, т. е. такой, которая позволит разработчику либо устранить соответствующую ошибку, либо классифицировать ее как ложное срабатывание, или же она является неполной и непригодной для ручного анализа. Помимо этого, полученные критические элементы будут в дальнейшем играть ключевую роль при визуализации — именно их необходимо показать пользователю для ручного анализа, а их отсутствие будет означать, что трасса непригодна для ручного анализа.

[1] Development of environment model specifications for static verification of Linux Kernel: http://linuxtesting.org/31-08-2020.

[2] http://linuxtesting.org/16-08-2016, http://linuxtesting.org/27-08-2017, http://linuxtesting.org/28-08-2017, http://linuxtesting.org/13-08-2018, http://linuxtesting.org/28-08-2020, http://linuxtesting.org/20-08-2021, http://linuxtesting.org/23-08-2021

В данной статье:

- формально определено понятие тщательности трассы ошибки в верификации программного обеспечения;
- оценена тщательность трасс ошибок для инструментальных средств, участвующих в соревнованиях по верификации программного обеспечения, на основе анализа их результатов;
- продемонстрировано, как тщательность трасс ошибок может использоваться в качестве метрики для сравнения инструментальных средств верификации, а также предложены практические рекомендации по повышению показателя валидации.

1. Определения

Авторы рассматривают программную систему, написанную на языке C, которую требуется проверить на соответствие определенным свойствам. Проверяемые **свойства** либо являются общими для языка C (такими как безопасность работы с памятью, отсутствие переполнений, отсутствие гонок данных и завершимость программы), либо выражаются как недостижимость (отсутствие достижимости) конкретной функции (например, `abort()` или `error()`).

Задача верификации включает в себя исходный код программы, компонент программной системы, подлежащий анализу, точку входа (например, функцию `main()`), а также список свойств для проверки. Кроме того, задача верификации может содержать ограничения по ресурсам и различные параметры конфигурации. **Верификатор** — это инструментальное средство, автоматически решающее поставленную задачу верификации и выдающее результат, указывающий, нарушается ли определенное свойство в программе или нет.

Результат верификации представляет собой формальный ответ на поставленную задачу верификации и обычно относится к одному из трех возможных исходов, перечисленных далее.

- **TRUE**. Верификатор доказал, что указанное свойство не может быть нарушено в данной программе, и предоставил **доказательство корректности**. Если доказательство оказывается ошибочным, такая ситуация называется **пропущенной ошибкой**.
- **FALSE**. Верификатор обнаружил нарушение свойства в программе и предоставил **трассу ошибки**. Если конкретная трасса ошибки не соответствует реальной ошибке в программе, это считается **ложным срабатыванием** инструмента.
- **UNKNOWN**. Верификатор не смог решить поставленную задачу верификации в силу нехватки ресурсов или ошибок в самом инструменте. При этом сгенерированный лог может быть полезен разработчикам верификатора как отчет об ошибке.

Трасса ошибки представляет собой один или несколько путей выполнения в исходном коде программы от точки входа до нарушения свойства. В свою очередь, **доказательство корректности** — это граф, описывающий все возможные выполнения программы от точки входа, не содержащие нарушения свойств [7]. В дальнейшем будут рассматриваться исключительно результаты верификации, соответствующие общему формату SV-COMP [6], принятому в соревнованиях среди верификаторов.

Валидатор — это инструментальное средство [8], которое получает на вход задачу верификации и соответствующий результат верификации и определяет, является ли предоставленный результат корректным. Процесс валидации направлен на повышение доверия к результатам верификации. При последующем анализе полученных результатов человек может учитывать информацию, предоставленную валидатором.

Процесс верификации проиллюстрирован на рисунке. Каждая задача решается верификатором, а полученные результаты затем могут проходить стадию валидации. Этот этап полностью автоматизирован и позволяет минимизировать затраты ресурсов за счет использования быстрых инструментальных средств и параллельного решения различных задач верификации с применением облачных технологий.

Так как валидация может закончиться неуспешно или может быть неточной, то появляется необходимость ручного анализа результатов. Основная задача этапа — классифицировать все найденные трассы ошибок на ложные срабатывания и реальные ошибки, которые затем могут быть устранены

Процесс верификации

разработчиками программного обеспечения. Для этого важно предоставлять результаты в формате, который будет понятен разработчикам. Помимо этого, полученные доказательства корректности должны быть проверены на предмет наличия пропущенных ошибок. Эта задача априори более сложная в силу структуры доказательств, и в данной работе она не будет рассмотрена. Таким образом, снижение требуемых ресурсов на анализ результатов верификации является достаточно актуальной задачей.

Данная работа ставит следующий исследовательский вопрос: какие именно элементы трасс ошибок являются критически важными для успешной валидации и в дальнейшем для более быстрого анализа результатов?

1.1. Бенчмарки SV-COMP

Бенчмарки соревнования SV-COMP сгруппированы в отдельные категории верификации, каждая из которых нацелена на проверку определенного свойства программного обеспечения. Свойства, включенные в SV-COMP'23, следующие:

- *unreach* проверяет, достижимы ли вызовы определенных функций в программе;
- *memory* гарантирует отсутствие ошибок, связанных с памятью, таких как разыменование нулевого указателя или использование памяти после освобождения;
- *concurrency* рассматривает проблемы многопоточных программ, такие как гонки по данным, взаимные блокировки и другие ошибки работы с потоками;
- *termination* определяет, завершается ли программа для всех возможных входных данных, предотвращая бесконечные циклы или бесконечную рекурсию;
- *overflow* проверяет, что арифметические операции не выходят за допустимые числовые границы, предотвращая непреднамеренное поведение, вызванное переполнением.

Каждая категория включает набор задач верификации, для которых заранее известен ожидаемый результат, что позволяет точно оценивать работу инструментальных средств. Задачи различаются по сложности и по предметной области — от общих тестов до специализированного программного обеспечения, например, драйверов Linux.

Каждый участвующий инструментарий пытается решить поставленные задачи в рамках заданных ограничений по ресурсам, выдавая результат верификации для каждой задачи. Этот результат должен совпадать с известным ожидаемым значением, а также успешно пройти независимую валидацию. Результаты соревнования дают представление о сильных и слабых сторонах каждого инструментального средства, способствуя развитию области верификации программного обеспечения.

Процесс валидации в SV-COMP хорошо отлажен — определенные верификаторы (такие как CPAchecker [9] или Ultimate Automizer [10]) с заданными конфигурациями принимают на вход исходный код и результат. Они подтверждают его корректность или отклоняют [8], при этом процесс требует значительных ресурсов. На основании результатов соревнований SV-COMP можно вычислить процент успешной валидации для каждого инструментального средства и для каждого проверяемого свойства — фактически это формальный показатель качества полученных результатов. Кроме того, текущий процесс валидации не предоставляет информации о причинах отклонения результатов или о том, каких элементов в нем могло не хватать.

Другим, более наглядным подходом к оценке результатов, является их визуализация. В настоящее время SV-COMP не предлагает стандартизированных рекомендаций по визуализации результатов, оставляя эту задачу на усмотрение отдельных верификаторов или сторонних инструментальных средств. Например, CPAchecker [9] поддерживает визуализацию внутренних представлений найденных трасс, однако такие представления могут быть трудны для восприятия сторонними разработчиками. В свою очередь, проекты LDV Tools [11] и Klever [5] предлагают более удобные средства визуализации, позволяющие пользователям вручную анализировать трассы ошибок, построенные определенными инструментальными средствами, однако эти проекты не универсальны и не могут быть применены к произвольным программам.

2. Тщательность трасс ошибок

В этом разделе рассмотрена структура трасс ошибок, на основе которой сформулировано определение их тщательности. Дан анализ наиболее значимых элементов трасс, влияющих на уровень тщательности в контексте свойств, используемых в SV-COMP.

2.1. Структура трассы ошибки

Для начала рассмотрим структуру результатов в формате GraphML [6]. Этот результат включает заголовок, содержащий вспомогательную информацию: путь к исходным файлам, описание свойства, язык исходного кода, использованный

инструментарий, временную метку и другие параметры. Помимо этого, он содержит набор узлов и ребер, формирующих граф, который представляет внутреннюю структуру трассы ошибки или доказательства корректности. В случае трассы ошибки граф описывает один или несколько возможных путей, ведущих к ошибке. Каждый узел может содержать дополнительные метки, к числу которых относятся перечисленные далее.

- Entry — начальный узел графа.
- Violation — указывает на нарушение свойства в трассе ошибки.
- Sink — обозначает необследованные пути в условиях внутри трассы ошибки.
- Invariant — задает инвариант в виде C-выражения и его область действия для доказательств корректности.

Ребра графа соответствуют операциям, связанным со строками исходного кода. Классифицируем все виды элементов на перечисленные далее группы.

- **Function Calls** (FC) (метки: *enterFunction*, *returnFromFunction*) — фундаментальный структурный элемент трассы ошибки, отражающий переходы управления между функциями. Такие элементы фиксируют вызов функции и возврат из нее, что позволяет восстановить последовательность вызовов, приведшую к нарушению свойства.
- **Conditions** (C) (метка *control*) показывают, какая ветка была выбрана: *condition-true* (истинная) или *condition-false* (ложная). Неисследованные условия могут вести к sink-узлам в трассах ошибок.
- **Assumptions** (A) (метка *assumption*) задают C-выражение, считающееся истинным в рамках данного блока. Используются для оценки значений переменных, аргументов и возвращаемых значений функций.
- **Thread specifics** (TS) (метка *threadId*) содержит идентификатор текущего потока для каждого ребра, а *createThread* указывает на создание нового потока.
- **Loop heads** (LH) (метка *enterLoopHead*) определяют и ограничивают переходы к операциям в точках входа в циклы в управляющем потоке программы.

Ребро может также содержать метку *sourcecode* с соответствующим исходным кодом или ссылкой на исходный код (метки *startline, endline, startoffset, endoffset*), что упрощает восстановление операций. Эти метки особенно полезны для пошагового воспроизведения ошибок и являются критичными для визуализации.

Важно подчеркнуть, что описанные элементы не являются универсально обязательными для проверки всех видов багов. Например, свойство *memory* требует операций, связанных с памятью: выделение, присваивание и освобождение (например, операции вида assumption). Более того, избыточное включение всех операций в трассу ошибки может затруднить ее восприятие, так как пользователю требуется только та информация, которая имеет отношение к найденному багу. Тем не менее в общем случае (особенно в сложных задачах, как в категории *SoftwareSystems* на SV-COMP) включение всех элементов может повысить шансы на успешную валидацию.

2.2. Формальное определение тщательности трасс ошибок

Рассмотрим множество верификационных задач T, а также соответствующие результаты верификации, полученные выбранным верификатором для свойства P (*unreach, memory, concurrency, termination или overflow*). Обозначим через W множество трасс ошибок, соответствующих данным результатам. **Процент успешной валидации**, обозначаемый как R, определяется следующим образом:

$$R = \frac{|\mathcal{W}_{validated}|}{|\mathcal{W}|} \times 100\,\%,$$

где $W_{validated} \subseteq W$ — подмножество успешно провалидированных трасс.

Как обсуждалось ранее, каждая трасса $w \in W$ содержит элементы, относящиеся к пяти различным категориям: FC, C, A, TS, LH. Трасса $w \in W$ считается содержащей элемент определенного типа, если в ней содержится более одного элемента этого типа. Если трасса содержит только один тривиальный элемент (например, единственный FC для точки входа в программу, совпадающую во всех инструментах), она не считается фактически содержащей этот тип.

Наличие элементов определенных категорий повышает шансы на успешную валидацию. Будем называть такие типы элементов *релевантными*. Пусть $\mathcal{E}_w \subseteq \{\text{FC, C, A, TS, LH}\}$ — множество типов элементов, содержащихся в трассе w, $\mathcal{E}_{relevant} \subseteq \{\text{FC, C, A, TS, LH}\}$ — множество релевантных типов элементов. Точные критерии определения множества релевантных типов элементов составляют основную задачу работы и будут рассмотрены в следующем подразд.

Тщательность для заданного множества трасс W, обозначаемая как $WT(W)$, определяется как максимальный процент релевантных типов элементов по трассам из W:

$$WT(W) = \max_{w \in W} \frac{|\mathcal{E}_w \cap \mathcal{E}_{relevant}|}{|\mathcal{E}_{relevant}|} \times 100\,\%.$$

Таким образом, тщательность трасс измеряет, насколько полно принятая модель включает релевантные элементы, необходимые для валидации. Так как не все верификационные задачи одинаково сложны (некоторые могут вообще не содержать условий или других важных элементов), рассматриваем трассу с наивысшим значением тщательности, чтобы простые задачи верификации не занижали оценку способности инструмента генерировать полные трассы. Нормализация по числу релевантных элементов делает эту метрику сопоставимой между различными свойствами.

Тщательность трасс ошибок является полезной метрикой для сравнения различных моделей оценки (средств их реализующих) по заданному свойству P и множеству задач T. Следует отметить, что модель тщательности зависит от свойства P, так как каждое свойство характеризуется своим набором релевантных элементов.

Тщательность показывает, насколько хорошо трасса способствует успешной валидации. Это делает метрику ценной для оценки эффективности инструментов при минимальных вычислительных затратах.

2.3. Оценка релевантных элементов для свойств SV-COMP

В этом подразделе оценим релевантные элементы для свойств SV-COMP, используя результаты соревнований [12] в качестве обучающего набора данных. В набор вошли трассы ошибок, полученные 43 верификаторами по пяти свойствам, вместе с соответствующими результатами валидации. Рассматривались все верификаторы, которые участвовали в соответствующих категориях SV-COMP'23 и выдали как минимум десять трасс ошибок. Полное описание процесса воспроизведения результатов содержится в артефакте [13].

Главный вопрос, подлежащий анализу и исследованию, заключался в следующем: какие типы элементов — FC, C, A, TS, LH — наиболее важны для валидации по каждому свойству? Для ответа на него была разработана модель линейной регрессии, чтобы проанализировать зависимость между этими пятью элементами трасс и процентом успешной валидации.

Для проведения этого анализа использовалась регрессия Automatic Relevance Determination (ARD-регрессия) [14]. ARD-регрессия — это разновидность байесовской регрессии, особенно полезная при работе с многомерными пространствами признаков. Благодаря использованию априорных распределений, стимулирующих разреженность, ARD автоматически определяет значимость каждого признака, позволяя модели взвешивать вклад каждого элемента трасс в процент успешной валидации. Этот подход удобен тем, что снижает риск переобучения и обеспечивает использование только наиболее информативных признаков, что повышает интерпретируемость и надежность результатов. Кроме того, он отдает предпочтение простым решениям — модель по умолчанию стремится использовать меньше признаков, если данные явно не указывают на их важность.

С помощью ARD-регрессии исследовалась взаимосвязь между элементами трасс и процентом успешной валидации по различным свойствам соревнований SV-COMP. Результаты, приведенные в табл. 1, показывают, какие элементы оказывают наибольшее влияние на успешную валидацию. Для наглядности были нормализованы коэффициенты и сфокусированы на положительных значениях более 0,3 [14], так как они указывают на прямую зависимость с процентом успешной валидации. Отрицательные коэффициенты не рассматривались, поскольку они свидетельствуют о противоположной связи — включение таких элементов снижает процент валидации, что не является целью анализа в контексте настоящего исследования. Следует отметить, что поставленную задачу можно было бы решить методом полного перебора всех возможных комбинаций элементов. Однако целью данного исследования являлось не достижение максимального значения тщательности, а выявление возможной зависимости между процентом успешной валидации и наличием релевантных элементов в трассе. Кроме того, предложенный подход масштабируем и может быть применен к случаям, когда число анализируемых элементов трассы значительно превышает пять.

Проделанный анализ также показывает, как можно провести аналогичные расчеты для любых

Таблица 1

Нормализованные коэффициенты ARD-регрессии, отражающие зависимость между каждым элементом трасс ошибок и процентом успешной валидации

Свойство	FC	TS	A	C	LH
Unreach	**1**	0	**0,95**	**1**	**1**
Memory	0,112	0,266	**1**	0,112	0
Concurrency	0	**0,356**	**1**	0	0
Termination	−0,202	0	**0,629**	**0,798**	0
Overflow	0	−0,089	**0,911**	0	0

Примечание. Выделены положительные коэффициенты более 0,3.

других свойств, для которых имеются размеченные трассы ошибок. Однако также наблюдается, что инструментальные средства с одинаковым уровнем тщательности могут генерировать разные трассы ошибки, что приводит к различным процентам успешной валидации. Таким образом, хотя тщательность трасс и является необходимым условием для достижения высокого процента валидации, она не является единственным определяющим фактором.

На основании этого анализа можно выделить следующие релевантные элементы для свойств SV-COMP:

- *unreach*: FC, A, C и LH;
- *memory*: A;
- *concurrency*: A и TS;
- *termination*: A и C;
- *overflow*: A.

3. Оценка тщательности трасс для верификаторов SV-COMP

В этом разделе дано определение тщательности трасс для всех инструментальных средств, участвовавших в SV-COMP'23 для верификации C-программ по каждому проверяемому свойству отдельно. Два верификатора (CoVeriTeam-Verifier-AlgoSelection и CoVeriTeam-Verifier-ParallelPortfolio) были исключены из сравнения, так как они состоят из других верификаторов, уже включенных в анализ. В данной оценке анализ проводится исключительно для трасс ошибок. Результаты с представленными элементами трасс ошибок, рассчитанной тщательностью и процентом валидации приведены в табл. 2—6 (релевантные столцы выделены).

Таблица 2

Сравнение тщательности трасс ошибок с процентом успешной валидации для свойства *unreach* в SV-COMP'23

Верификатор	**FC**	TS	**A**	**C**	**LH**	Валидация, %	Тщательность, %
Bubaak	**0**	0	**1**	**0**	**0**	62,07	25
CBMC	**0**	1	**1**	**0**	**0**	5,61	25
CPA-BAM-BnB	**1**	0	**1**	**1**	**1**	83,1	100
CPA-BAM-SMG	**1**	0	**1**	**1**	**1**	85,07	100
CPAChecker	**1**	1	**1**	**1**	**1**	89,58	100
Crux	**0**	0	**1**	**0**	**0**	0,13	25
ESBMC-kind	**0**	1	**1**	**0**	**0**	21,74	25
Graves-CPA	**1**	1	**1**	**1**	**1**	84,51	100
Graves-Par	**1**	1	**1**	**1**	**1**	93,33	100
Infer	**0**	0	**0**	**0**	**0**	0	0
PeSCo-CPA	**1**	1	**1**	**1**	**1**	80,28	100

Таблица 3

Сравнение тщательности трасс ошибок с процентом успешной валидации для свойства *memory* в SV-COMP'23

Верификатор	FC	TS	**A**	C	LH	Валидация, %	Тщательность, %
2LS	0	0	**1**	0	0	61,29	100
Bubaak	0	0	**1**	0	0	97,94	100
CBMC	0	1	**1**	0	0	67,89	100
CPAChecker	1	1	**1**	1	1	96,26	100
DIVINE	0	0	**1**	0	0	0	100
ESBMC-kind	0	1	**1**	0	0	90,47	100
Graves-CPA	1	1	**1**	1	1	95,7	100
Graves-Par	1	1	**1**	1	1	95,89	100
PeSCo-CPA	1	1	**1**	1	1	99,32	100
Symbiotic	0	1	**1**	0	1	98,85	100
UAutomizer	1	1	**1**	1	1	94,58	100
Ukojak	1	0	**1**	1	0	96,06	100
UTaipan	1	1	**1**	1	0	94,32	100

Таблица 4

Сравнение тщательности трасс ошибок с процентом успешной валидации для свойства *concurrency* в SV-COMP'23

Верификатор	FC	TS	A	C	LH	Валидация, %	Тщательность, %
CBMC	0	**1**	**1**	0	0	84,81	100
CPA-Lockator	1	**1**	**1**	1	1	26,51	100
CPAChecker	1	**1**	**1**	1	1	100	100
Cseq	1	**1**	**1**	0	0	25,89	100
Dartagnan	0	**1**	**0**	0	0	90,24	50
Deagle	0	**1**	**1**	0	0	88,71	100
DIVINE	0	**0**	**1**	0	0	80,87	50
EBF	0	**0**	**0**	0	0	83,43	0
ESBMC-incr	0	**1**	**1**	0	0	79,41	100
ESBMC-kind	0	**1**	**1**	0	0	89,73	100
Graves-CPA	1	**1**	**1**	1	1	99,23	100
Graves-Par	1	**1**	**1**	1	1	100	100
Infer	0	**0**	**0**	0	0	0	0
Lazy-CSeq	1	**1**	**1**	1	0	94,89	100
LF-checker	0	**1**	**1**	0	0	85,31	100
PeSCo-CPA	1	**1**	**1**	1	1	100	100
PIChecker	1	**0**	**1**	1	1	98,14	50
Symbiotic	0	**1**	**1**	0	1	92,73	100
UAutomizer	1	**1**	**1**	1	1	94,68	100
UgemCutter	1	**1**	**1**	1	0	96,47	100
UTaipan	1	**1**	**1**	1	0	95,76	100

Таблица 5

Сравнение тщательности трасс ошибок с процентом успешной валидации для свойства *termination* в SV-COMP'23

Верификатор	FC	TS	**A**	**C**	LH	Валидация, %	Тщательность, %
2LS	0	0	**1**	**0**	0	69,08	50
Bubaak	0	0	**1**	**0**	0	34,78	50
CPAChecker	1	1	**1**	**1**	1	97,01	100
Graves-CPA	1	1	**1**	**1**	1	81,15	100
Pesco-CPA	1	1	**1**	**1**	1	98,26	100
Symbiotic	0	1	**1**	**0**	1	52,96	50
UAutomizer	1	1	**1**	**1**	1	98,24	100
VeriFuzz	0	0	**0**	**1**	1	71,34	50

Таблица 6

Сравнение тщательности трасс ошибок с процентом успешной валидации для свойства *overflow* в SV-COMP'23

Верификатор	FC	TS	A	C	LH	Валидация, %	Тщательность, %
2LS	0	0	1	0	0	95,7	100
Bubaak	0	0	1	0	0	94,67	100
CBMC	0	1	1	0	0	62,14	100
CPAChecker	1	1	1	1	1	100	100
Crux	0	0	1	0	0	95,05	100
ESBMC-kind	0	1	1	0	0	66,69	100
Frama-C-SV	0	0	0	0	0	0	0
Graves-Par	1	1	1	1	1	2	100
Infer	0	0	0	0	0	0	0
Pinaka	0	0	1	0	0	100	100
Symbiotic	0	1	1	0	1	100	100
UAutomizer	1	1	1	1	1	100	100
UKojak	1	0	1	1	0	100	100
UTaipan	1	1	1	1	0	100	100
VeriFuzz	0	0	1	0	1	90,81	100

Проведенная оценка показывает, что тщательность трасс в целом коррелирует с более высоким процентом успешной валидации для большинства свойств. Однако есть примечательные исключения, например, EBF [15] для свойства *concurrency*, где достигается высокий процент успешной валидации даже при низкой тщательности.

Разные свойства демонстрируют различную степень зависимости от тщательности трасс. Например, такие свойства, как *unreach и termination*, показывают прямую зависимость от тщательности, в то время как другие, такие как *memory и concurrency*, достигают достаточного процента валидации даже с низкой тщательностью. При этом есть и исключения из правил, например, Graves-CPA [16] при максимальной тщательности демонстрирует высокий процент валидации для всех свойств, кроме *overflow*. Этот факт свидетельствует о том, что в этих случаях важнее наличие конкретных элементов трасс ошибок (например, определенного условия), нежели просто наличия некоторого типа элементов.

Инструментальные средства, такие как CPAchecker [9] и Pesco-CPA [17], стабильно показывают хорошие результаты по всем свойствам, достигая как высокой тщательности трасс, так и высокого процента успешной валидации. Их стабильная эффективность подчеркивает их пригодность в качестве надежных решений для произвольных задач верификации программного обеспечения. Напротив, инструментальные средства с низкой тщательностью, такие как Crux [18] и Infer [19], демонстрируют низкий процент успешной валидации по большинству свойств из-за пустых трасс ошибок.

Из данных представленных таблиц можно сделать вывод о том, какие элементы трасс ошибок следует добавить для повышения процента успешной валидации конкретных инструментальных средств. Например, хотя CBMC [20] показывает отличные результаты для свойства *concurrency* (84,81 % валидации), он значительно отстает на более сложных задачах *unreach* в силу отсутствия в трассах таких элементов, как FC и C. Аналогично, Symbiotic [21] превосходно справляется с задачами *memory*, *concurrency и overflow* (100 % тщательности), но показывает слабые результаты на задачах *termination* (50 % тщательности) в силу отсутствия необходимых условий (C).

Таким образом, данное исследование подтверждает возможность достаточно точной аппроксимации процента успешной валидации с помощью тщательности. Тем не менее авторы также отмечают, что инструментальные средства с одинаковыми значениями тщательности трасс ошибок могут формировать трассы с различными конкретными элементами данного типа, что приводит к разным результатам при валидации. Этот результат подчеркивает тот факт, что хоть тщательность и является необходимым условием для достижения высокого процента успешной валидации, в целом этого недостаточно.

3.1. Анализ тщательности трасс ошибок для победителей SV-COMP

Еще один вопрос, требующий исследования, который возникает: действительно ли победители всегда демонстрируют наивысшую тщательность? В табл. 7—11 дан сравнительный анализ тщательности трасс ошибок среди победителей по всем свойствам SV-COMP'23 (номер перед названием инструментария соответствует месту, которое было им занято в соответствующей категории в соревнованиях SV-COMP'23).

Для таких свойств, как *memory*, *overflow и concurrency*, наблюдается прямая зависимость —

Таблица 7

Сравнение тщательности трасс ошибок у победителей SV-COMP'23 для свойства *unreach* (в категории *SoftwareSystems*)

Верификатор	FC	A	C	LH	Тщательность, %
I. Symbiotic	0	1	0	1	50
II. Bubaak	0	1	0	0	25
III. Mopsa	Нет трасс				0

Таблица 8

Сравнение тщательности трасс ошибок у победителей SV-COMP'23 для свойства *memory*

Верификатор	A	Тщательность, %
I. Symbiotic	1	100
II. CPAChecker	1	100
III. UTaipan	1	100

Таблица 9

Сравнение тщательности трасс ошибок у победителей SV-COMP'23 для свойства *concurrency*

Верификатор	TS	A	Тщательность, %
I. Deagle	1	1	100
II. Uautomizer	1	1	100
III. UgemCutter	1	1	100

Таблица 10

Сравнение тщательности трасс ошибок у победителей SV-COMP'23 для свойства *termination*

Верификатор	A	C	Тщательность, %
I. VeriFuzz	1	0	50
II. Uautomizer	1	1	100
III. 2LS	1	0	50

Таблица 11

Сравнение тщательности трасс ошибок у победителей SV-COMP'23 для свойства *overflow*

Верификатор	A	Тщательность, %
I. Uautomizer	1	100
II. Ukojak	1	100
III. Utaipan	1	100

победители стабильно показывают как высокие проценты успешной валидации, так и максимальной тщательности трасс. Однако для свойств *unreach и termination* подобная закономерность не прослеживается. Победители в этих категориях зачастую достигают лишь умеренной или минимальной тщательности (например, инструментарий Mopsa [22] с 0 %-ной тщательностью для *unreach*).

Более детальный анализ показывает, что такие инструментальные средства набирают высокие баллы за счет генерации большего числа валидированных доказательств корректности. Например, Mopsa не выдал ни одной трассы ошибки, тогда как Symbiotic [21] предоставил лишь пять трасс ошибок для *unreach*. Поскольку данное исследование сосредоточено исключительно на трассах ошибок, используемый принцип подсчета очков мог повлиять на результаты.

Еще один важный момент заключается в том, что валидация существенно зависит от конкретного набора трасс ошибок — если этот набор слишком прост для проверки, полученный процент успешной валидации может не отражать реальную эффективность инструментария. Это обстоятельство указывает на то, что процент валидации сам по себе не является универсально надежным критерием для оценки производительности инструментального средства по всем задачам в рамках одного свойства.

Отмеченные выше выводы подчеркивают ценность введенной метрики тщательности, которая формально оценивает качество трасс ошибок. Включение подобной метрики в SV-COMP могло бы восполнить существующие пробелы в оценке потенциальной пользы от полученных результатов верификации.

Заключение

Формально определено понятие тщательности трасс ошибок в верификации программного обеспечения, которое основано на выявлении ключевых элементов, релевантных проверяемому свойству. С использованием бенчмарков из соревнования по верификации программного обеспечения SV-COMP проведено детальное исследование этих ключевых элементов и получены оценки тщательности трасс всех инструментальных средств, участвующих в соревнованиях. Результаты представленного анализа показывают четкую зависимость между тщательностью трасс ошибок и процентом успешной валидации результатов верификации. Этот факт показывает, как тщательность может служить практической метрикой для сравнения средств верификации и предлагать новый критерий для оценки качества трасс ошибок. Кроме того, по результатам исследования предложены конкретные рекомендации по повышению тщательности, а значит, и процента успешной валидации для каждого инструментального средства и проверяемого свойства.

В будущем предполагается, что работа будет сосредоточена на визуализации трасс ошибок как практическом приложении метрики тщательности. Представление ключевых элементов трассы в ясном и лаконичном виде крайне важно для эффективного ручного анализа. Метрика тщательности может служить ориентиром для поиска баланса, выделяя критические элементы, важные для анализа ошибки, и исключая второстепенные детали, которые, наоборот, не важны для проверяемого свойства. Такой подход станет важным шагом на пути к созданию более доступных и интерпретируемых результатов верификации в удобном для ручного анализа формате.

Дополнительно планируется расширить концепцию тщательности трасс ошибок на доказательства корректности. В отличие от трасс ошибок, которые имеют линейную последовательную структуру, доказательства представляют собой нелинейные графы, что существенно усложняет их анализ. Такое расширение потребует разработки дополнительных критериев оценки качества, позволяющих определить, имеет ли место пропущенная ошибка.

Список литературы

1. **Mordan V., Novikov E.** Minimizing the number of static verifier traces to reduce time for finding bugs in Linux kernel modules // Proc. SYRCoSE'2014. 2014. DOI: 10.15514/SYRCOSE-2014-8-5.

2. **Beyer D.** Software verification: 10th comparative evaluation (SV-COMP 2021) // Proc. TACAS 2021. 2021. Vol. 2. P. 401—422. DOI: 10.1007/978-3-030-72013-1_24.

3. **Beyer D.** Progress on software verification: SV-COMP 2022 // Proc. TACAS 2022. 2022. Vol. 2. P. 375—402. DOI: 10.1007/978-3-030-99527-0_20.

4. **Beyer D.** Competition on software verification and witness validation: SV-COMP 2023 // Proc. TACAS 2023. 2023. Vol. 2. P. 495—522. DOI: 10.1007/978-3-031-30820-8_29.

5. **Novikov E., Zakharov I.** Verification of operating system monolithic kernels without extensions // *Proc. ISoLA 2018*. 2018. Vol. IV. P. 230—248. DOI: 10.1007/978-3-030-03427-6_19.

6. **Beyer D., Dangl M., Dietsch D.** et al. Verification witnesses // *Proc. ACM* 2022. 2022. Vol. 31, No. 4. Article 57. DOI: 10.1145/3477579.

7. **Beyer D., Dangl M., Dietsch D., Heizmann M.** Correctness witnesses: exchanging verification results between verifiers // Proc. 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016. P. 326—337. DOI: 10.1145/2950290.2950351.

8. **Beyer D., Dangl M., Dietsch D.** et al. Witness validation and stepwise testification across software verifiers // Proc. 10th Joint Meeting on Foundations of Software Engineering. 2015. P. 721—733. DOI: 10.1145/2786805.2786867.

9. **Beyer D., Keremoglu M. E.** CPAchecker: A tool for configurable software verification // Proc. Computer Aided Verification. Berlin: Springer, 2011. P. 184—190. DOI: 10.1007/978-3-642-22110-1_16.

10. **Heizmann M., Christ J., Dietsch D.** et al. Ultimate Automizer with SMTInterpol (competition contribution) // Proc. TACAS 2013. 2023. P. 641—643. DOI: 10.1007/978-3-642-36742-7_53.

11. **Khoroshilov A., Mutilin V., Novikov E.** et al. Towards an open framework for C verification tools benchmarking // Perspectives of Systems Informatics. Berlin: Springer, 2012. P. 179—192. DOI: 10.1007/978-3-642-29709-0_17.

12. **Beyer D.** Verification Witnesses from Verification Tools (SV-COMP 2023) // Zenodo. 2023. DOI: 10.5281/zenodo.7627791.

13. **Mordan V.** Анализ структуры витнесов и их тщательности для SV-COMP'2023 // Zenodo. 2025. DOI: 10.5281/zenodo.15808896.

14. **Muller P., West M., MacEachern S.** Bayesian models for non-linear autoregressions // Journal of Time Series Analysis. 1997. Vol. 18, No. 6. P. 593—614. DOI: 10.1111/1467-9892.00070.

15. **Aljaafari F., Shmarov F., Manino E.** et al. EBF 4.2: Black-box cooperative verification for concurrent programs (competition contribution) // Proc. TACAS 2023. 2023. Vol. 2. P. 541—546. DOI: 10.1007/978-3-031-30820-8_33.

16. **Leeson W., Dwyer M.** Graves-CPA: A graph-attention verifier selector (competition contribution) // Proc. TACAS 2022. 2022. Vol. 2. P. 440—445. DOI: 10.1007/978-3-030-99527-0_28.

17. **Richter C., Wehrheim H.** PeSCo: Predicting sequential combinations of verifiers (competition contribution) // *Proc. TACAS* 2019. Vol. 3. P. 229—233. DOI: 10.1007/978-3-030-17502-3_19.

18. **Scott R., Dockins R., Ravitch T., Tomb A.** Crux: symbolic execution meets SMT-based verification (competition contribution) // Zenodo. Февраль 2022. DOI: 10.5281/zenodo.6147218.

19. **Kettl M., Lemberger T.** The static analyzer Infer in SV-COMP (competition contribution) // Proc. TACAS 2022. 2022. Vol. 2. P. 451—456. DOI: 10.1007/978-3-030-99527-0_30.

20. **Kröning D., Tautschnig M.** CBMC: C bounded model checker (competition contribution) // Proc. TACAS 2014. 2014. P. 389—391. DOI: 10.1007/978-3-642-54862-8_26.

21. **Ayaziová P., Strejček J.** Symbiotic-Witch 2: More efficient algorithm and witness refutation (competition contribution) // Proc. TACAS 2023. 2023. Vol. 2. P. 523—528. DOI: 10.1007/978-3-031-30820-8_30.

22. **Monat R., Ouadjaout A., Miné A.** Mopsa-C: Modular domains and relational abstract interpretation for C programs (competition contribution) // Proc. TACAS 2023. 2023. Vol. 2. P. 565—570. DOI: 10.1007/978-3-031-30820-8_37.

Evaluating Key Elements of Error Traces in Static Software Verification

V. O. Mordan, Research Associate, mordan@ispras.ru,
Ivannikov Institute for System Programming of the Russian Academy of Sciences, Moscow, 109004, Russian Federation,
V. S. Mutilin, Leading Researcher, mutilin@ispras.ru,
Ivannikov Institute for System Programming of the Russian Academy of Sciences, Moscow, 109004, Russian Federation;
Moscow Institute of Physics and Technology (State University), Dolgoprudny, Moscow Oblast, 141701, Russian Federation

Corresponding author:

Vitalii O. Mordan, Research Associate, Ph.D. (Physics and Mathematics),
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
Moscow, 109004, Russian Federation
E-mail: mordan@ispras.ru

Received on May 29, 2025
Accepted on July 08, 2025

Software verification is a resource-intensive process that combines automatic program analysis with manual results analysis. While recent advancements in verification techniques and cloud-based infrastructures have significantly accelerated the automatic phase, manual analysis remains a critical bottleneck—particularly for large-scale software systems—due to the need for specialized developer expertise. Verification results typically include warnings about potential bugs, which must either be corrected or classified as false positives.

In this work, we address this challenge by introducing a formal notion of thoroughness for error traces in the context of software verification. We define thoroughness as a measure of how comprehensively an error trace captures essential elements necessary for understanding and validating a reported issue. To evaluate this concept in practice, we apply it to the verification results produced by tools participating in the Software Verification Competition (SV-COMP). Our analysis demonstrates that the proposed metric is not only feasible but also informative in real-world verification scenarios.

Furthermore, we argue that thoroughness can serve as a meaningful metric for comparing verification tools, identifying strengths and weaknesses in how they represent potential errors. It also aids in improving automated validation processes. In addition, our evaluation shows how the metric can be used to determine which specific elements must be present in an error trace to support effective visualization. By identifying these key elements, we aim to facilitate more efficient manual analysis and promote the development of more user-friendly verification tools.

Keywords: *software verification, thoroughness, verification results, validation, error trace*

For citation: **Mordan V. O., Mutilin V. S.** Evaluating Key Elements of Error Traces in Static Software Verification, *Programmnaya Ingeneria*, 2025, vol. 16, no. 11, pp. 558-569. DOI: 10.17587/prin.16.558-569.

References

1. **Mordan V., Novikov E.** Minimizing the Number of Static Verifier Traces to Reduce Time for Finding Bugs in Linux Kernel Modules, *SYRCoSE'2014*, 2014. DOI: 10.15514/SYRCOSE-2014-8-5.

2. **Beyer D.** Software Verification: 10th Comparative Evaluation (SV-COMP 2021), *Proc. TACAS 2021*, 2021, vol. 2, pp. 401—422. DOI: 10.1007/978-3-030-72013-1_24.

3. **Beyer D.** Progress on Software Verification: SV-COMP 2022, *Proc. TACAS 2022*, 2022, vol. 2, pp. 375—402. DOI: 10.1007/978-3-030-99527-0_20.

4. **Beyer D.** Competition on Software Verification and Witness Validation: SV-COMP 2023, *Proc. TACAS* 2023, 2023, vol. 2, pp. 495—522. DOI: 10.1007/978-3-031-30820-8_29.

5. **Novikov E., Zakharov I.** Verification of Operating System Monolithic Kernels Without Extensions, *Leveraging Applications of Formal Methods, Verification and Validation. ISoLA 2018,* 2018, vol. IV, pp. 230—248. DOI: 10.1007/978-3-030-03427-6_19.

6. **Beyer D., Dangl M., Dietsch D.** et al. Verification witnesses, *Proc. ACM 2022*, 2022, vol. 31, no. 4, article 57. DOI: 10.1145/3477579.

7. **Beyer D., Dangl M., Dietsch D., Heizmann M.** Correctness witnesses: exchanging verification results between verifiers, *Proc. 24th ACM SIGSOFT Int. Symp. Foundations of Software Engineering*, 2016, pp. 326—337. DOI: 10.1145/2950290.2950351.

8. **Beyer D., Dangl M., Dietsch D.** et al. Witness validation and stepwise testification across software verifiers, *Proc. 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 721—733. DOI: 10.1145/2786805.2786867.

9. **Beyer D., Keremoglu M. E.** CPAchecker: A Tool for Configurable Software Verification, *Proc. Computer Aided Verification*, Berlin, Springer, 2011, pp. 184—190. DOI: 10.1007/978-3-642-22110-1_16.

10. **Heizmann M., Christ J., Dietsch D.** et al. Ultimate Automizer with SMTInterpol (Competition Contribution), *Proc. TACAS 2013,* 2013, pp. 641—643. DOI: 10.1007/978-3-642-36742-7_53.

11. **Khoroshilov A., Mutilin V., Novikov E.** et al. Towards an Open Framework for C Verification Tools Benchmarking, *Perspectives of Systems Informatics*, Berlin, Springer, 2012, pp. 179—192. DOI: 10.1007/978-3-642-29709-0_17.

12. **Beyer D.** Verification Witnesses from Verification Tools (SV-COMP 2023), *Zenodo*, 2023. DOI: 10.5281/zenodo.7627791.

13. **Mordan V.** Witness Structure and Thoroughness Analysis for SV-COMP'2023, *Zenodo*, 2023. DOI: 10.5281/zenodo.15808896 (in Russian).

14. **Muller P., West M., MacEachern S.** Bayesian Models for Non-linear Autoregressions, *Journal of Time Series Analysis*, 1997, vol. 18, no. 6, pp. 593—614. DOI: 10.1111/1467-9892.00070.

15. **Aljaafari F., Shmarov F., Manino E.** et al. EBF 4.2: Black-Box Cooperative Verification for Concurrent Programs (Competition Contribution), *Proc. TACAS 2023*, 2023, vol. 2, pp. 541—546. DOI: 10.1007/978-3-031-30820-8_33.

16. **Leeson W., Dwyer M.** Graves-CPA: A Graph-Attention Verifier Selector (Competition Contribution), *Proc. TACAS 2022,* 2022, vol. *2*, pp. 440—445. DOI: 10.1007/978-3-030-99527-0_28.

17. **Richter C., Wehrheim H.** PeSCo: Predicting Sequential Combinations of Verifiers (Competition Contribution), *Proc. TACAS 2019*, 2019, vol. 3, pp. 229—233. DOI: 10.1007/978-3-030-17502-3_19.

18. **Scott R., Dockins R., Ravitch T., Tomb A.** Crux: Symbolic execution meets SMT-based verification (Competition Contribution), *Zenodo*, 2022. DOI: 10.5281/zenodo.6147218.

19. **Kettl M., Lemberger T.** The Static Analyzer Infer in SV-COMP (Competition Contribution), *Proc. TACAS 2022*, 2022, vol. 2, pp. 451—456. DOI: 10.1007/978-3-030-99527-0_30.

20. **Kröning D., Tautschnig M.** CBMC: C Bounded Model Checker (Competition Contribution), *Proc. TACAS 2014,* 2014, pp. 389—391. DOI: 10.1007/978-3-642-54862-8_26.

21. **Ayaziová P., Strejček J.** Symbiotic-Witch 2: More Efficient Algorithm and Witness Refutation (Competition Contribution), *Proc. TACAS 2023*, 2023, vol. 2, pp. 523—528. DOI: 10.1007/978-3-031-30820-8_30.

22. **Monat R., Ouadjaout A., Miné A.** Mopsa-C: Modular Domains and Relational Abstract Interpretation for C Programs (Competition Contribution), *Proc. TACAS 2023*, 2023, vol. 2, pp. 565—570. DOI: 10.1007/978-3-031-30820-8_37.