DOI: 10.15514/ISPRAS-2024-36(4)-13

Проблема неопределенности в анализе трасс на основе высокоуровневых моделей в контексте динамической верификации

А.А. Карнов, ORCID: 0000-0002-2066-9946 <karnov@ispras.ru>

Институт системного программирования РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В данной статье обсуждается проблема применения метода динамической верификации к большим и сложным системам, в частности, операционным системам общего назначения. Современные практики и стандарты разработки критических систем требуют наличия формальной модели политики безопасности. Полнота и непротиворечивость формальных требований, указанных в модели политики безопасности, должна быть верифицирована с использованием формальных методов. Позже, когда будет разработана реализация системы, необходимо установить, что реализованные механизмы безопасности соответствуют указанным в модели требованиям. При использовании такого подхода удобно иметь единую модель, подходящую как для формальной верификации, так и для тестирования реализации. Для достижения этой цели необходимо, с одной стороны, выделить подмножество языковых конструкций модели, подходящих для обоих методов, а с другой, разработать специальные методики анализа трасс выполнения, позволяющие эффективно выполнять тысячи тестов. В статье приводится анализ языковых конструкций, позволяющих эффективное использование модели в рамках динамической верификации. Также в статье представлены методы оптимизации процесса динамической верификации систем. Предложенные методы были реализованы в прототипе инструмента анализа трасс и протестированы на модели системы контроля доступа для операционных систем на основе Linux.

Ключевые слова: динамическая верификация; анализ трасс; язык описания моделей Event-B.

Для цитирования: Карнов А. А. Проблема неопределенности в анализе трасс на основе высокоуровневых моделей в контексте динамической верификации. Труды ИСП РАН, том 36, вып. 4, 2024 г., стр. 169–182. DOI: 10.15514/ISPRAS–2024–36(4)–13.

Uncertainty Problem in High-Level Model-Based Trace Analysis as Part of Runtime Verification

A.A. Karnov, ORCID: 0000-0002-2066-9946 <karnov@ispras.ru>

Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. The article discusses the problem of applying runtime verification to large and complex systems such as general-purpose operating systems. When verifying the security mechanisms of operating systems, modern practices and standards require a formal security policy model (SPM). The SPM must be verified using formal model methods, and it must also be used to verify the completeness and consistency of the operating system's security mechanisms by confirming compliance with the formal requirements of the SPM. In this case, it is convenient to have a single model suitable for both formal verification and implementation testing. For practical application, it is necessary, on the one hand, to select a subset of model language constructs suitable for both acts, and on the other hand, to develop special techniques for analyzing execution traces that allow to effectively perform thousands of test cases. The article addresses both of these issues. We present an analysis of language constructs that allow us to use the model for both verification and execution trace analysis. We also offer techniques that have been developed to optimize the runtime verification of Linux-based systems. We also implemented the proposed methods in the trace analysis tool prototype.

Keywords: runtime verification; trace analysis; Event-B modelling language.

For citation: Karnov A.A. Uncertainty problem in high-level model-based trace analysis as part of runtime verification. *Trudy ISP RAN/Proc. ISP RAS*, vol. 36, issue 4, 2024. pp. 169-182 (in Russian). DOI: 10.15514/ISPRAS-2024-36(4)-13

1. Введение

Динамическая верификация [1] – это подход к анализу и исполнению вычислительных систем, основанный на извлечении из системы информации во время ее работы и последующем анализе полученной информации. Информация извлекается в виде последовательности событий в порядке их возникновения в системе, при этом события могут дополняться данными, например, параметрами и неким результатом. Такая последовательность событий называется трассой выполнения [2] и представляет собой наблюдаемое поведение системы. Анализ трассы показывает, является ли наблюдаемое поведение корректным, иначе говоря, соответствует ли оно определенным свойствам или же нарушает их.

Динамическую верификацию также можно определить как набор формальных методов для установления соответствия между трассой выполнения и спецификацией системы. В статье рассматривается частный случай, когда в роли такой спецификации выступает формальная модель. Как правило, модель системы содержит меньшее количество деталей и имеет меньший объем по сравнению с реализацией. Кроме того, одной модели может соответствовать сразу несколько различных реализаций. По этим причинам формальная верификация модели системы является более доступной и выгодной по сравнению с формальной верификацией самой системы. Использование верифицированной модели, как источника определения корректного поведения системы, значительно повышает надежность динамической верификации.

Область приведенного в этой статье исследования ограничена анализом трасс выполнения и формальной моделью. Мы абстрагируемся от других важных компонентов динамической верификации, оставляя за рамками исследования методики инструментирования и тестирования системы. Реализация является лишь источником трасс, которые могут быть собраны как во время запусков неких тестовых сценариев, так и во время штатной работы системы. Этапы сбора и извлечения информации также не рассматриваются.

Исследование направлено на эффективное проведение анализа трасс на основе формальной модели системы и на преодоление проблем, с которыми придется неизбежно столкнуться. Раздел 2 посвящен мотивации выбора используемых инструментов и подходов, в нем также перечисляются цели исследования. Раздел 3 содержит краткое описание проблемы на простом примере. В разделе 4 представлен краткий обзор существующих подходов к анализу трасс и исполнению формальных моделей. Текущие результаты исследования представлены в разделе 5. В разделе 6 перечислены будущие направления работы.

2. Мотивация

Для работы над критически важными системами, такими как средства защиты информации, современные стандарты и практики требуют [3] наличия формальной модели политики безопасности. Модель политики безопасности должна быть верифицирована [4] при помощи формальных методов, например, дедуктивной верификации. Возможность использовать одну и ту же формальную модель как для формальной спецификации критической системы, так и для ее тестирования является одним из важнейших преимуществ динамической верификации. Используя такой подход, мы получаем возможность построить замкнутую систему верификации, в которой корректность наблюдаемого поведения системы будет сопоставляться с верифицированной моделью политики безопасности.

Именно по этой причине нам необходимо представить формальную модель тестируемой системы в форме, удобной как для методов формальной верификации, так и для динамической верификации. Оказывается, предложить такое представление на практике достаточно сложно. Одно из возможных решений предложено в данной статье.

Основное практическое применение предложенного подхода – верификация средств защиты информации операционных систем. В этой области было проделано уже достаточное количество работ [5], [6], [7]. В качестве основного средства описания формальных моделей был выбран язык Event-B. Event-B имеет набор преимуществ: простые и понятные языковые конструкции, Rodin [8] – удобную среду разработки моделей, включающую инструменты дедуктивной верификации и множество других полезных инструментов. Среди инструментов есть инструменты для выполнения (анимации) моделей, что демонстрирует принципиальную возможность использования языка для анализа трасс.

Нет никаких серьезных причин отказываться от Event-B, но существуют нетривиальные проблемы, которые усложняют, а иногда и не позволяют использовать модели для динамической верификации. Цели данного исследования:

1. Провести анализ и классификацию языковых конструкций и приемов написания спецификаций на Event-B, усложняющих динамическую верификацию;
2. Выделить подмножество языка и шаблоны написания фрагментов спецификаций, которые либо устраняют проблемы анализа трасс, либо позволяют привести модели к виду, удобному для динамической верификации;
3. Разработать набор техник трансформации моделей;
4. Разработать методы анализа моделей.

3. Проблема неопределенности

Рассмотрим простую модель светофоров на пешеходном переходе. Состояние модели состоит из двух логических переменных:

$$cars\_go \in BOOL \quad (1)$$

$$peds\_go \in BOOL \quad (2)$$

Для обеспечения безопасности движения модель содержит инвариант, запрещающий одновременное движение автомобилей и пешеходов:

$$\neg(cars\_go = TRUE \wedge peds\_go = TRUE) \quad (3)$$

В качестве начального состояния мы можем взять любое состояние, не нарушающее инвариант 3. События $cars$ и $peds$ могут изменять значение переменных на значение параметра go. Чтобы гарантировать, что инвариант 3 не будет нарушен, события имеют условия безопасности, которые называются охранными условиями. Охранное условие 4 относится к событию $cars$, а охранное условие 5 относится к $peds$.

$$go = TRUE \Rightarrow peds\_go = FALSE \quad (4)$$

$$go = TRUE \Rightarrow cars\_go = FALSE \quad (5)$$

Пространство состояний модели показано на рис. 1.

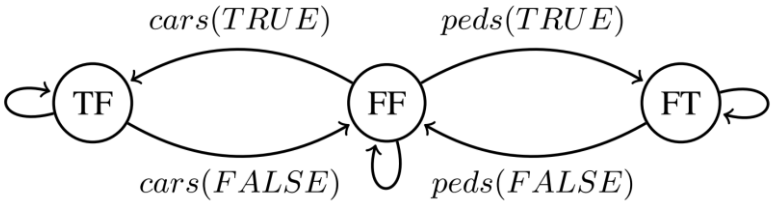

Рис. 1. Пространство состояний модели светофоров.
Fig. 1. Traffic lights state space.

Такая модель не подходит для тестирования, ведь мы не можем напрямую получить информацию о том, двигаются ли пешеходы и автомобили. Мы можем отслеживать только сигналы светофора, поэтому нам необходимо добавить контекст модели. Контекст будет содержать константы red и $green$, обозначающие разные цвета сигналов:

$$red \in COLORS \quad (6)$$

$$green \in COLORS \quad (7)$$

$$red \neq green \quad (8)$$

Чтобы получить возможность реагировать на сигналы светофора, в уже существующие события нужно добавить новый параметр $colors$. Этот параметр соответствует множеству загоревшихся на светофоре сигналов, что обозначено в охранном условии 9. При этом параметр go в трассе будет отсутствовать, и вывод о его значении приходится делать из охранных условий 10–11. Охранные условия 9–11 нужно добавить в оба события.

$$colors \subseteq COLORS \quad (9)$$

$$green \in colors \Rightarrow go = TRUE \quad (10)$$

$$green \notin colors \Rightarrow go = FALSE \quad (11)$$

В данной модели наличие зеленого света разрешает движение независимо от того, какие еще сигналы горят в этот момент на светофоре. Несмотря на это неудобное для пешеходов и водителей допущение, модель является полной и непротиворечивой. Но при попытке ее исполнить мы неизбежно столкнемся со следующими сложностями.

Рассмотрим трассу $\{cars(\{green\}),\ cars(\{red\}),\ peds(\{green\})\}$. Трасса соответствует модели только в том случае, если в начальном состоянии переменная $peds\_go$ имеет значение $FALSE$: состояния FF и TF на рис. 1. Но мы также допустили, что начальное состояние может быть FT. С этой ситуацией мы столкнемся в рамках Event-B, если будем

использовать действия с неопределенными присваиваниями. Всего в языке существует три вида присваиваний: простое присваивание, произвольный выбор значения из множества возможных значений и произвольный выбор значения, удовлетворяющего предикату. Так как произвольный выбор значения по предикату – единственный способ компактно записать конструкцию if-then-else, нежелательно ограничиваться только простым присваиванием и не рассматривать неопределенные присваивания вообще.

Если мы посмотрим на данные с реального светофора, мы сможем наблюдать события $cars(\{yellow\})$, $cars(\{red,\ yellow\})$ и $cars(\emptyset)$. Значение множества $COLORS$ следует из набора предикатов логики первого порядка 6–8. Эти предикаты вполне допускают наличие дополнительных цветов на светофоре, так что значение множества, как совокупности всех его элементов, остается неопределенным. Это важная проблема, так как предикаты – единственный способ определить значение констант в Event-B, кроме того, существуют и другие обстоятельства, когда значение нужно вывести из предиката. Иногда существует только одно решение, удовлетворяющее предикату, но для множества $COLORS$ из примера существует бесконечное количество решений. Кроме того, модель не содержит упомянутый в трассе идентификатор $yellow$. С точки зрения модели желтый вполне может оказаться не дополнительным цветом, а тем же самым красным (тогда участникам движения надо стоять), зеленым (тогда нужно двигаться) или вообще не быть цветом (тогда в трассе кроется ошибка). Эта проблема важна, так как на практике мы не можем описать каждую сущность, например, каждый файл или пользователя в модели в виде отдельной константы.

Как уже упоминалось, параметр go в трассе будет отсутствовать. Эта проблема имеет корни в выразительных возможностях языка Event-B. Единственный способ вычислить некоторое промежуточное значение – добавить еще один параметр к событию и ограничить его значение через охранные условия. Такой параметр мы называем вычислимым. Очевидное решение проблемы, не использовать вычислимых параметров вовсе, ведет к ухудшению читаемости модели. Есть и еще одна сторона данной проблемы. С точки зрения синтаксиса модели случай, когда трасса не является полной из-за ошибки на этапе сбора информации, не отличим от случая, когда значение параметра нужно вычислить из охранных условий.

Возможность абстрагироваться от лишних деталей является основным преимуществом моделирования. Например, в модели светофоров мы абстрагировались от наличия дополнительных цветов и оставили множество $COLORS$ неопределенным полностью. Подобный прием в теории может понадобиться и для других типов данных, например, для чисел. Тогда во время анализа трассы важное свойство должно быть проверено без использования точного значения. Но это также может оказаться спорной ситуацией: абстракция неотличима от случайной неточности в спецификации.

4. Другие работы

Набор инструментов ProB [9] включает в себя инструмент, называемый аниматором. Аниматор позволяет выполнять события Event-B модели, значения параметров при этом ограничиваются или точно определяются при помощи дополнительных предикатов. Ядро ProB написано на языке SICStus Prolog, поэтому ProB активно использует возможности логического программирования для того, чтобы подобрать подходящее значение для каждого объекта. На вид логических формул при этом не накладывается никаких формальных ограничений. Аниматор может дать пользователю возможность самому выбрать определенное значение, если от этого зависит следующее состояние модели. Трассы аниматора нелинейны и могут содержать ветви в разные состояния. К сожалению, использовать аниматор для больших тестовых случаев невозможно, так как он не может работать с большими наборами значений, длинными трассами и большим количеством переменных. Это является следствием выбранного метода и приоритетов разработчиков ProB: поддержка всех конструкций языка важнее, чем эффективность.

Существуют работы по комбинированию SMT-солверов [10-11] с Event-B моделями для их исполнения. SMT-солверы активно используются на отдельных частях модели в процессе дедуктивной верификации, но при исполнении всей модели целиком они сталкиваются со значительными трудностями. Не все солверы поддерживают теорию множеств. Также солверы не могут поддерживать предикаты над таким объектом, как множество всех подмножеств конечного множества. В этом случае формула, задающая объект, выходит за рамки логики первого порядка [12]. Таким образом, SMT-солверы могут быть использованы не на всей модели и, как и логические программы, имеют проблемы с временной эффективностью.

Наиболее эффективным с точки зрения времени выполнения является подход генерации кода [13]. Event-B модели транслируются [14-16] в исполняемую программу на императивном языке, которая может быть запущена. Несложно передать такой программе на вход трассу и обработать ее. Но существующие инструменты генерации кода не рассматривают проблему неопределенности: точные значения объектов, в том числе констант, должны передаваться в конфигурационных файлах.

Таким образом, не существует эффективного решения, покрывающего все потребности тестирования на основе формальной Event-B модели.

5. Текущие результаты

5.1 Пользовательские типы

В Event-B объекты могут иметь логический тип (как элементы встроенного булевого множества), целочисленный тип (как элементы встроенного целочисленного множества), пользовательский тип (как элементы определяемого пользователем множества, называемого в Event-B несущим множеством), являться упорядоченной парой (элемент декартова произведения двух множеств) или множеством (как элемент множества всех подмножеств базового типа). Event-B нотации всех типов приведены в табл. 1. Примером пользовательского типа является $COLORS$ в разделе 3. Объекты пользовательских типов не имеют никакого явного значения и могут быть связаны с другими объектами того же типа только через отношения равенства и неравенства. Примером такого отношения может быть предикат 8. Неопределенность значения следует уже из природы этих объектов.

В качестве значения такого объекта можно взять пару из имени его несущего множества (типа) и некоторого натурального числа. В табл. 2 показан пример интерпретации несложного набора предикатов. Метод выбора целого числа рассматривается в следующем подразделе.

Табл.1. Типы объектов Event-B.
Table 1. Event-B object types.

Тип	**Нотация Event-B**
Логический	$x \in BOOL$
Целочисленный	$x \in \mathbb{Z}$
Пользовательский	$x \in A$
Упорядоченная пара	$x \in A \times B$
Множество	$x \in \mathbb{P}(A)$

Этот подход довольно похож на тот, что используется в ProB. Множества (в том числе несущие) представлены в ProB в виде списков без повторяющихся элементов. Значением

объекта пользовательского типа является конкатенация из имени несущего множества и индекса элемента в соответствующем списке. В примере из таб. 2 символьные значения для x и y будут $A1$ и $A2$, соответственно. Данная нотация удобна для подстановки в трассу новых объектов, не объявленных в модели в качестве констант.

Табл.2. Интерпретация пользовательских типов.
Table 2. Interpretation of user-given type.

Event-B	**Интерпретация**	**Решение**
$x \in A$ $y \in A$ $z \in A$ $x \neq y$ $x = z$	$GIVEN(A,i) \in A$ $GIVEN(A,j) \in A$ $GIVEN(A,k) \in A$ $GIVEN(A,i) \neq GIVEN(A,j)$ $GIVEN(A,i) = GIVEN(A,k)$ $x = GIVEN(A,i)$ $y = GIVEN(A,j)$ $z = GIVEN(A,k)$	$i = 1$ $j = 2$ $k = 1$

Предложение A.1. *Для всех объектов пользовательских типов, значение которых должно явно отличаться от значений заданных в модели констант, в трассе используется идентификатор, состоящий из имени типа и целого числа.*

Альтернативный подход широко используется при генерации кода, когда вместо несущего множества можно использовать тип данных. Это значительно упрощает работу с большими несущими множествами, так как не требует хранения в памяти всех возможных значений. Но это может усложнять обработку логических формул: проверка принадлежности несущему множеству превращается в проверку типа данных, а выражения над несущими множествами (например, «все цвета, кроме красного») необходимо переписывать в другой форме. Разумеется, возможен и описанный выше подход с обычными множествами.

В SMT-солверах также можно использовать оба подхода. Можно считать, что все объекты пользовательского типа являются строкой и использовать в качестве значения таких объектов реальные данные (например, для объекта типа «пользователь» значением будет имя пользователя в системе). Это упрощает процесс тестирования, не требуя отображения имен. Несущим множеством будет множество из всех строк. Также можно создать тип данных (так называемый сорт) и значение объекта будет комбинацией из типа и целого числа.

5.2 Вычисление значения из предиката

В Event-B объекты могут обозначаться идентификаторами, например, константы и переменные. В случае переменных значение объекта определяется через действие присваивания. Но константы и вычисляемые параметры событий определяются иначе: их значение необходимо вывести из набора предикатов.

Значение объекта, на которое указывает идентификатор, может быть точно определено, если оно задается простым предикатом равенства. Из предикатов 12–14 легко извлечь точные значения объектов.

$$NUMBER = 20 \tag{12}$$

$$2 \mapsto TRUE = PAIR \tag{13}$$

$$SET = \{A, B\} \tag{14}$$

Но в некоторых случаях предикаты ведут к неопределенным значениям. Из предиката 15 следует, что $NUMBER$ – любое число, меньшее либо равное 20. Предикат 16 указывает, что $PAIR$ – любая упорядоченная пара из множества REL, представляющего из себя некое

отображение. В предикатах 17–18 множество SET содержит элементы A, B и может содержать любые другие элементы того же типа.

$$NUMBER \leq 20 \tag{15}$$

$$PAIR \in REL \tag{16}$$

$$A \in SET \tag{17}$$

$$B \in SET \tag{18}$$

Далее мы называем предикаты, из которых делается вывод о значении объекта, определяющими. Если определяющий предикат является равенством, в котором один из операндов – идентификатор объекта, то второй операнд мы называем определяющим выражением. Для спецификаций систем, подходящих для тестирования, мы должны ограничить виды определяющих предикатов, чтобы избежать неопределенности.

Предложение B.1. *Для всех логических и целочисленных объектов, а также для упорядоченных пар, значение которых следует из предикатов, определяющий предикат должен иметь форму равенства между идентификатором объекта и определяющим выражением.* Так, для $NUMBER$ и $PAIR$ мы допускаем определяющие предикаты 12–13 и запрещаем определяющие предикаты 15–16. Определяющие выражения могут содержать другие идентификаторы, пока все идентификаторы точно определены и в их определениях нет циклов:

$$NUMBER \mapsto TRUE = PAIR \tag{19}$$

$$NUMBER = 2 \tag{20}$$

Предложение B.2. *Для каждого объекта пользовательского типа должно быть задано отношение равенства или неравенство со всеми остальными объектами того же типа.* Так как у объектов пользовательского типа в рамках модели отсутствует какое-либо значение, необходимо знать, во-первых, какие объекты равны друг другу, во-вторых, для всех остальных объектов необходимо явное указание на их различие. В примере из табл. 2 это требование соблюдено. В таком случае мы можем пронумеровать все объекты, учитывая отношения равенства. Именно так конструируется решение, указанное в последнем столбце табл. 2. В дополнение к равенствам и неравенствам, в современных версиях Event-B существует предикат раздела (partition). Он позволяет выразить неравенство объектов в более компактной форме:

$$partition(S, S1, S2) \Leftrightarrow (S1 \cup S2 = S) \wedge (S1 \cap S2 = \emptyset) \tag{21}$$

Если несущее множество S содержит четыре константы A, B, C и D, гораздо удобнее записать в формате раздела, чем прописывать шесть отдельных неравенств для каждой пары из A, B, C и D:

$$partition(S, \{A\}, \{B\}, \{C\}, \{D\}) \tag{22}$$

Множества также могут быть определены через предикат равенства, но в некоторых случаях это невозможно. Для такого определения требуется перечислить все элементы множества, при этом для каждого элемента необходимо ввести значение, а в случае пользовательского типа данных – идентификатор соответствующей константы. Если мощность множества велика, это неудобно, если вообще возможно. Поэтому необходимо добавить альтернативный способ.

Предложение B.3. *Определяющие предикаты для множеств должны быть предикатами равенства из Предложения B.1, либо предикатами из табл. 3.* Набор определяющих предикатов может быть рассмотрен, как задача теории множеств. Для этой задачи будет существовать общее решение, но при этом точное значение множества останется неопределенным. Предполагая изначально, что множество SET является пустым, мы можем обработать предикаты и построить минимальное частное решение. Все остальные предикаты,

в которых фигурирует множество SET, могут быть использованы, чтобы убедится, что минимальное решения является корректным в рамках модели. Эту же логику можно применить к отображениям и функциям, которые могут быть рассмотрены, как множества упорядоченных пар.

Табл.3. Определяющие предикаты для множеств.
Table 3. Defining predicates for sets.

Предикат	**Решение**
$x \in SET$	$\{x\} \subseteq SET \subseteq \Omega^*$
$s \subseteq SET$	$s \subseteq SET \subseteq \Omega$
$partition(SET, s1, s2)$	$SET = s1 \cup s2$
$partition(s1, SET, s2)$	$SET = s1 \setminus s2$

* Ω обозначает множество всех объектов того же типа, что и элементы SET.

Минимальное решение может оказаться некорректным из-за предикатов, не входящих в множество определяющих. Такой случай проиллюстрирован предикатами 23–24. Необходимо дополнить множества S и REM новыми элементами, чтобы выполнить условие из предиката 24.

$$partition(S, \{A\}, \{B\}, REM) \tag{23}$$

$$REM \neq \emptyset \tag{24}$$

Также необходимо полностью заполнить множества, которые фигурируют в присваиваниях: отсутствие даже одного элемента в таких множествах приведет к некорректному состоянию модели. Для того чтобы дополнить множество, необходимо знать его точную мощность и факт, что оно конечно.

Предложение B.4. *Все множества, появляющиеся в присваиваниях или требующих дополнения для выполнения аксиом, должны быть конечными и их мощность должна быть точно определена.* Примером необходимого определения является предикат 25.

$$finite(SET) \wedge card(SET) = 20 \tag{25}$$

Во всех остальных случаях в качестве значения можно оперировать минимальным решением, динамически расширяя его по мере того, как в трассе появляются новые элементы.

В настоящее время мы выбираем, дополнить ли множество новыми элементами заранее или расширять его динамически уже в ходе анализа трассы, по тому, определена ли его точная мощность. Но этот подход не учитывает распространенный случай, когда мощность указана точно, но при этом велика, а в предварительном дополнении множества нет никакой необходимости. В таком случае динамическое расширение этого множества полезно для повышения эффективности. Одна из будущих целей этого исследования – реализация алгоритма, который может обнаруживать такие множества и оптимизировать вычисления над ними.

5.3 Неопределенные присваивания

В Event-B существует три вида действий присваивания: простое присваивание, произвольный выбор из множества возможных значений, произвольный выбор значения, удовлетворяющего предикату. Из всех видов только простое присваивание лишено неопределенности. В качестве примеров можно рассмотреть присваивания 26–31, которые полностью аналогичны определяющим предикатам 12–18.

$$NUMBER \coloneqq 20 \tag{26}$$

$$PAIR \coloneqq 2 \mapsto TRUE \tag{27}$$

$$SET \coloneqq \{A, B\} \tag{28}$$

$$NUMBER :| NUMBER' \leq 20 \tag{29}$$

$$PAIR :\in REL \tag{30}$$

$$SET :| A \in SET' \wedge B \in SET' \tag{31}$$

Произвольный выбор из множества значений будет лишен неопределенности только в случае, когда множество выбора содержит одно и только одно значение. В таком виде данный вид присваивания полностью лишен смысла. В случае неопределенности алгоритм анализа, тем не менее, понятен: мы должны проанализировать остаток трассы многократно, каждый раз выбирая следующее из возможных значений. Если произошла некая ошибка, это означает, что возможно, выбор был некорректным и в тестируемой системе он был сделан иначе. Необходимо вернуться в точку выбора и изменить его.

Несмотря на прозрачность процесса, долгие возвраты и экспоненциальный рост количества комбинаций с каждым новым выбором делают такой анализ очень неэффективным. Кроме того, если мы рассматриваем тестирование, то подобные неточности в спецификации неизбежно вызовут вопросы. По этим причинам мы не допускаем произвольный выбор из множества в любой его форме.

Предложение C.1. *Произвольный выбор из множества значений недопустим.*

Присваивание значения, удовлетворяющего предикату, в большинстве случаев также является неопределенным. При этом такое присваивание – единственный способ реализации конструкции if-then-else в рамках Event-B. Поэтому вместо запрета необходимо сформулировать ограничения. Если сформулирован предикат p, соответствующий условию, и две альтернативы A и B, соответствующие итоговым значениям переменной в зависимости от выполнения условия p, то присваивание можно записать следующим образом:

$$var \coloneqq p \wedge var' = A \vee \neg p \wedge var' = B \tag{32}$$

Эту форму можно расширить для любого числа условий, пока выполняются следующие условия. Во-первых, дизъюнкция всех условий должна быть тождественна истине (условие полноты). Полнота гарантирует, что всегда найдется хотя бы одно итоговое значение для переменной. Во-вторых, конъюнкции всех возможных пар условий должны быть невыполнимы. Это условие определенности, гарантирующее, что возможно только одно итоговое значение. В примере выше условием полноты будет формула $p \vee \neg p \Leftrightarrow \top$, а условием определенности $p \wedge \neg p \Leftrightarrow \bot$. Оба условия будут выполнены.

При этом в событии, которое содержит присваивание, могут быть охранные условия. Предположим, что охранное условие g – единственное. В таком случае мы можем ослабить условие полноты и не рассматривать те случаи, которые заведомо противоречат охранному условию. Тогда присваивание и условие полноты примут следующий вид:

$$var \coloneqq p \wedge var' = A \vee q \wedge var' = B \tag{33}$$

$$(g \Rightarrow p \vee q) \Leftrightarrow \top \tag{34}$$

Продолжая эти рассуждения, мы можем, наконец, сформулировать общий случай для n альтернативных значений и k охранных условий.

Предложение C.2. *Присваивание значения, удовлетворяющего предикату, допустимо, если оно имеет вид 35, удовлетворяет условию полноты 36 и условию определенности 37. Любые другие формы данного вида присваивания запрещены.*

$$var : | \bigvee_{i=1}^{n} (p_i \wedge var' = Val_i) \tag{35}$$

$$((\bigwedge_{j=1}^{k} g_j) \Rightarrow (\bigvee_{i=1}^{n} p_i)) \Leftrightarrow \top \tag{36}$$

$$\forall i, j \cdot 0 < i < j \leq n: p_i \wedge p_j \Leftrightarrow \bot \tag{37}$$

5.4 Отсутствующие параметры

Как упоминалось в разделе 3, значения некоторых параметров, осмысленных в рамках модели, могут отсутствовать в тестируемой системе и должны быть вычислены из охранных условий. С точки зрения анализа трассы эти параметры попросту отсутствуют, что усложняет автоматическое вынесение вердикта в этих случаях.

Предложение D.1. *Если значение параметра может отсутствовать в трассе, оно должно быть однозначно вычислимо из охранных условий.*

В этом случае, как и прежде, требуем точных значений этих параметров, но не требуем формул специального вида. В настоящее время мы решаем эту проблему при помощи медиатора – компонента, который переводит трассу из набора собранных в тестируемой системе данных в термины модельных сущностей. Этот компонент специфичен для каждой пары из модели и реализации. Медиатор позволяет провести анализ трассы, абстрагируясь от конкретной реализации системы. На этот же компонент можно переложить задачу по вычислению и добавлению в трассу отсутствующих ранее параметров.

Предложение D.2. *Значения всех вычислимых параметров должны быть вычислены до анализа корректности трассы.*

Снова рассмотрим пример со светофорами из раздела 3. Трасса выполнения подается на вход медиатору. Медиатор должен подставить в трассу характерные для модели идентификаторы red и $green$, если на светофоре зажглись соответствующие сигналы. Если зажегся желтый сигнал, то в соответствии с Предложением A.1 в трассу будет подставлен идентификатор $COLORS3$. Также, если зажегся зеленый сигнал, то значение параметра go станет $TRUE$ согласно охранному условию 10, в противном случае в трассе окажется значение $FALSE$ согласно охранному условию 11.

Необходимо учитывать, что таким образом мы добавляем в трассу свойства, которые отсутствуют в наблюдаемом поведении. При неточной реализации медиатора это может сказаться на корректности вынесенного тестового вердикта. Этот подход усложняет разработку медиатора и может породить новые ошибки в процессе тестирования.

С другой стороны, такой подход увеличивает эффективность анализа и учитывает природу вычислимых параметров. При таком подходе вычислимый параметр явно отличается от просто отсутствующего, что помогает обнаруживать ошибки этапа сбора трассы.

Предложение D.3. *Если при анализе трассы охранное условие не может быть проверено из-за отсутствующего параметра, тест не пройден.* Данное решение сужает класс корректных трасс, поэтому для некоторых целей анализа оно может быть спорным.

Автоматическое вычисление отсутствующих параметров является целью дальнейших исследований.

5.5 Оптимизации

Для тестирования данного подхода был разработан прототип инструмента анализа трасс. В качестве языка программирования был выбран Java, так как для этого языка уже реализован набор библиотек для работы с Event-B моделями, что удобно для прототипа. Реализация,

использующая предложенный подход и делающая непосредственные вычисления всех объектов показала лишь небольшое ускорение по сравнению с аниматором ProB. Тем не менее, выяснилось, что прототип, не смотря на простейшую реализацию, все еще работал быстрее. Для достижения требуемых показателей эффективности анализа было необходимо реализовать некоторые оптимизации.

Предложение E.1. *Множества всех множеств, декартово произведение, множества отображений и функций должны иметь представление, отличное от обычных множеств.* В целях оптимизации стоит отказаться от хранения таких множеств в памяти и ввести для них дополнительное представление. Для этого также необходимо интерпретировать некоторые формулы иначе. Например, предикат 38 может быть переписан в более простой форме 39. Трансформация формул происходит автоматически.

$$SUBSET \in \mathbb{P}(SET) \tag{38}$$

$$SUBSET \subseteq SET \tag{39}$$

Предложение E.2. *При обработке кванторов необходим автоматический анализ формулы, нацеленный на ограничение области значений подкванторных переменных.* Измерения времени работы инструмента показали, что решение кванторных формул является наиболее затратной по времени задачей. Если добавить в анализатор дополнительные правила обработки таких формул, ограничивая множество перебираемых значений, то обработка формулы ускоряется на порядок.

Предложение E.3. *Проверка инвариантов должна быть опциональной.* Инварианты являются большим набором предикатов, который нужно проверять после выполнения каждого события в трассе. Инварианты часто содержат кванторы, так что даже с оптимизациями их проверка занимает значительную часть времени анализа.

При этом если анализ трассы проводится на основе ранее верифицированной модели, в проверке инвариантов нет никакой необходимости. Для верифицированной модели выполнение инвариантов логически следует из выполнения охранных условий. Все, что действительно требуется для проверки корректности состояния – проверить выполнение охранных условий. Разумеется, при работе с произвольными моделями этого недостаточно, но для повышения эффективности необходима опция отключения проверки инвариантов. В ProB такая возможность отсутствует.

Если модель не верифицирована, возможный способ оптимизации – проверять только те инварианты, которые содержат измененные последним событием переменные.

6. Будущие исследования

В настоящий момент все трансформации модели выполняются автоматически на уровне индивидуальных формул. Не было предложено техник трансформации моделей вручную в целях повышения эффективности тестирования. Необходимо провести более обширный анализ существующих практик создания формальных моделей. В настоящий момент основным ориентиром является модель контроля доступа и потока информации в ОС Linux [5, 17], но требуются эксперименты и с другими моделями.

Ограничения, предложенные в подразделе 5.2, являются строгими, но разумными, если речь идет о моделях, используемых в процессе тестирования. В случае, когда эти ограничения все же невозможно соблюдать, возможным решением может быть использование SMT-солверов на определенных частях модели. Например, это может оказаться полезно для подбора значений констант, так как в рамках одной модели это достаточно сделать один раз и в это никак не повлияет на скорость последующего анализа трасс. Необходимо провести эксперименты и оценить эффективность такого подхода.

Текущая версия инструмента не поддерживает отложенных вычислений, если не считать динамически расширяемых множеств. В примере из раздела 3 вполне возможно определить

неопределенное начальное состояние модели по первым событиям. Этот подход может быть использован, чтобы достичь в моделях большего уровня абстракции.

Желаемый результат данного исследования – достичь скорости выполнения тестов, максимально близкой к генерации кода. Тестирование прототипа инструмента анализа трасс на модели системы безопасности ОС Linux и 39000 тестовых трассах показало, что все трассы проверяются за 8 минут по сравнению с десятичасовым анализом с использованием ProB. Созданный вручную на языке Python исполняемый аналог модели позволил выполнить тот же набор тестов за 1 минуту.

7. Заключение

Данная статья посвящена исследованию проблем, которые возникают в процессе тестирования формальных систем, таких как системы безопасности операционных систем общего назначения, в частности, Linux. При этом необходимо построить формальную модель политики безопасности для того, чтобы определить требования к системе и иметь возможность доказать их непротиворечивость и полноту. Верифицированная модель политики безопасности используется при тестировании реализации системы для анализа корректности наблюдаемого поведения. Но этот анализ требует большого количества ресурсов и в некоторых случаях встречается с непреодолимыми препятствиями.

Это исследование выявило конкретные проблемы, связанные с неопределенностью значений. Мы обозначили подмножество языка Event-B, позволяющее избежать неопределенности и предложили подход для вычисления определенных значений и эффективного анализа трасс. Результаты исследования могут быть применены к другим формальным языкам моделирования, использующим логику первого порядка.

Также мы реализовали предложенный метод в прототипе инструмента анализа трасс. Прототип был протестирован на большом наборе тестовых трасс и реальной модели системы безопасности и показал значительно ускорение по сравнению с аниматором ProB. Этот анализ может быть выполнен непрерывно в рамках одного пространства состояний модели, что демонстрирует поддержку инструментом длинных тестовых трасс.

Список литературы / References

[1]. Bartocci E., Falcone Y., Fracalanza A., Reger G. Introduction to runtime verification. In Bartocci E., Falcone Y. (editors) Lectures on Runtime Verification. Lecture Notes in Computer Science, vol. 10457, Springer, 2018. pp. 1–33. DOI: 10.1007/978-3-319-75632-5_1.

[2]. Reger G., Havelund K. What Is a Trace? A Runtime Verification Perspective. In Margaria T., Steffen B. (editors) Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications. Lecture Notes in Computer Science, vol. 9953, Springer, 2016. DOI: 10.1007/978-3-319-47169-3_25.

[3]. ГОСТ Р 59453.1. Защита информации. Формальная модель управления доступом. Часть 1. Общие положения: описание стандарта и тендеры. – Введен 2021-06-01 – Москва: Стандартинформ, 2021. / State Std. GOST R 59453.1. Information protection. Formal access control model. Part 1. General principles. Moscow, 2021 (in Russian).

[4]. ГОСТ Р 59453.2. Защита информации. Формальная модель управления доступом. Часть 2. Рекомендации по верификации формальной модели управления доступом. – Введен 2021-06-01 – Москва: Стандартинформ, 2021. / State Std. GOST R 59453.1. Information protection. Recommendations on verification of formal access control model. Part 2. General principles. Moscow, 2021 (in Russian).

[5]. Девянин П. Н., Ефремов Д. В., Кулямин В. В., Петренко А. К., Хорошилов А. В., Щепетков И. В. Моделирование и верификация политик безопасности управления доступом в операционных системах. Горячая линия – Телеком, Москва, Россия, 2019. / Devyanin P.N., Efremov D. V., Kulyamin V. V., Petrenko A. K., Khoroshilov A. V., Shchepetkov I. V. Modeling and verification of access control security policies in operating systems. Moscow, Russia: Hotline-Telecom, 2019 (in Russian).

[6]. Ефремов Д. В., Копач В. В., Корныхин Е. В., Кулямин В. В. , Петренко А. К., Хорошилов А. В., Щепетков И. В. Мониторинг и тестирование модулей операционных систем на основе абстрактных моделей поведения системы. Труды ИСП РАН, 2021, том 33, вып. 6, стр. 15–26. DOI: 10.15514/ISPRAS-2021-33(6)-2. / Efremov D. V., Kopach V. V., Kornykhin E. V., Kulyamin V. V., Petrenko A.K., Khoroshilov A. V., Shchepetkov I. V. Runtime verification of operating systems based on abstract models. Proceedings of ISP RAS, 2021, vol. 33, no. 6, pp. 15–26 (in Russian). DOI: 10.15514/ISPRAS-2021-33(6)-2.

[7]. Девянин П. Н., Леонова М. А. Приёмы описания модели управления доступом ОССН Astra Linux Special Edition на формализованном языке метода Event-B для обеспечения её верификации инструментами Rodin и ProB. ПДМ, 2021, № 52, стр. 83–96. DOI: 10.17223/20710410/52/5. / Devyanin P. N., Leonova M. A. The techniques of formalization of OS Astra Linux Special Edition access control model using Event-B formal method for verification using Rodin and ProB. Prikladnaya Diskretnaya Matematica, 2021, no. 52, pp. 83–96, (in Russian). DOI: 10.17223/20710410/52/5.

[8]. Abrial J.-R., Butler M., Hallerstede S., Hoang T., Mehta F., Voisin L. Rodin: an open toolset for modelling and reasoning in Event-B. The International Journal on Software Tools for Technology Transfer, vol. 12. Springer, 2010, pp. 447–466. DOI: 10.1007/s10009-010-0145-y.

[9]. Leuschel M., Butler M. ProB: an automated analysis toolset for the B method. The International Journal on Software Tools for Technology Transfer, vol. 10, Springer, 2008, pp. 185–203. DOI: 10.1007/s10009-007-0063-9.

[10]. Déharbe D. Integration of SMT-solvers in B and Event-B development environments. Science of Computer Programming, vol. 78, 2013, pp. 310– 326. DOI: 10.1016/j.scico.2011.03.007.

[11]. Schmidt J., Leuschel M. SMT solving for the validation of B and Event-B models. The International Journal on Software Tools for Technology Transfer, vol. 24, Springer, 2022, pp. 1043–1077. DOI: 10.1007/s10009-022-00682-y.

[12]. Brauer E. Second-order logic and the power set. Journal of Philosophical Logic, vol. 47, 2018, pp. 123–142. DOI: 10.1007/s10992-016-9422-x.

[13]. Fürst A., Hoang T., Basin D., Desai K., Sato N., Miyazaki K. Code generation for Event-B. Presented at the Integrated Formal Methods 2014, Bertinoro, Italy, 09 2014.

[14]. Wright S., Automatic generation of C from Event-B. Presented at the Workshop on Integration of Model-based Formal Methods and Tools, Bangkok, Thailand, 02 2009.

[15]. Yang F., Jacquot J.-P., Souquières J. JeB: Safe simulation of Event-B models. Presented at the The 20th Asia-Pacific Software Engineering Conference, Bangkok, Thailand, 12 2013. DOI: 10.1109/APSEC.2013.83

[16]. Cataño N., Rivera V., EventB2Java: A code generator for Event-B. In Rayadurgam, S., Tkachuk, O. (editors) NASA Formal Methods. Lecture Notes in Computer Science, vol. 9690, Springer, 2016. DOI: 10.1007/978-3-319-40648-0_13.

[17]. Девянин П. Н. Модели безопасности компьютерных систем. Управление доступом и информационными потоками. Горячая линия – Телеком, Москва, Россия, 2013. / Devyanin P. N. Security models of computer systems. Control for access and information flows. Moscow, Russia: Hotline-Telecom, 2013, (in Russian).

Информация об авторах / Information about authors

Алексей Александрович КАРНОВ — аспирант. Научные интересы: формальные спецификации, верификация и тестирование, статический и динамический анализ.

Aleksei Aleksandrovich KARNOV — postgraduate student. Research interests: formal specifications, verification and testing, static and dynamic analysis.