DOI: 10.15514/ISPRAS-2025-37(6)-54

Генерация кода исполняемой модели Event-B на языке Python

[1,3] *А.А. Карнов, ORCID: 0000-0002-2066-9946 <karnov@ispras.ru>*
[1,2] *Е.В. Корныхин, ORCID: 0000-0001-9303-3132 <kornevgen@ispras.ru>*
[1] *Институт системного программирования имени В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1.*
[3] *НИУ Высшая школа экономики, 101978, Россия, г. Москва, ул. Мясницкая, д. 20.*

Аннотация. Данное исследование проводится в контексте динамической верификации средств защиты информации на основе формальной модели. Процесс верификации можно разбить на два этапа: формальная верификация модели и тестирование реализации с использованием верифицированной модели. Так как многие модели создаются с расчетом на формальную верификацию, для проведения первого этапа уже существуют готовые решения. При этом для тестирования реализации необходимо воспроизвести поведение модели, что является сложной задачей, особенно если рассматривать промышленное применение такого подхода. В статье предлагается возможная методика воспроизведения поведения модели при помощи сгенерированного на основе модели программного кода. Предлагаемая методика нацелена, с одной стороны, на получение эффективного по времени процесса воспроизведения поведения модели, а с другой, на увеличение читаемости программного кода и на визуальное соответствие между кодом и текстом исходной модели. Прозрачность процесса является важным качеством при работе с ответственными системами, а его простота дает возможность для широкого применения технологии на практике.

Ключевые слова: язык моделирования Event-B; язык программирования Python; генерация кода; динамическая верификация; анализ трасс; тестирование на основе моделей.

Для цитирования: Карнов А.А., Корныхин Е.В. Генерация кода исполняемой модели Event-B на языке Python. Труды ИСП РАН, том 37, вып. 6, часть 4, 2025 г., стр. 119–138. DOI: 10.15514/ISPRAS–2025–37(6)–54.

Generating Event-B Executable Model in Python

[1,3]*A.A. Karnov, ORCID: 0000-0002-2066-9946 <karnov@ispras.ru>*
[1,2]*E.V. Kornykhin, ORCID: 0000-0001-9303-3132 <kornevgen@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia.*

[3] *National Research University, Higher School of Economics, 20, Myasnitskaya ulitsa, Moscow, 101978, Russia.*

Abstract. The context of the research is a process of runtime verification of information security tools based on a formal model. The verification process can be divided into two stages: formal verification of a model and testing the implementation with use of the verified model. Since many models are created with formal verification in mind, there are many solutions for the first stage. At the same time, to test the implementation, it is necessary to reproduce the model behavior. This is a complex task, especially when considering the industrial application of this approach. The article proposes a possible technique for reproducing the model behavior using program code generated based on the model. The proposed technique is aimed, on the one hand, at obtaining a time-efficient process of reproducing the model behavior, and on the other hand, at increasing the readability of the program code and the visual correspondence between the code and the text of the original model. Transparency of the process is an important quality when working with critical systems, and simplicity allows for the wide application of the technology in practice.

Keywords: Event-B; Python; code generation; runtime verification; trace analysis; model-based testing.

For citation: Karnov A.A., Kornykhin E.V. Generating Event-B executable model in Python. Trudy ISP RAN/Proc. *ISP RAS*, vol. 37, issue 6, part 4, 2025. pp. 119-138 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-54.

1. Введение

Ответственные системы, такие как средства защиты информации (СЗИ), нуждаются в доказательстве их корректности. Корректность означает, что, во-первых, внутри системы должны выполняться некоторых свойства безопасности, а во-вторых, сами свойства полны и непротиворечивы. С 2021 года поэтапно разрабатывается группа стандартов ГОСТ Р 59453 «Защита информации. Формальная модель управления доступом» [1-4]. Данные стандарты предлагают технологию верификации СЗИ на основе формальных моделей. Требования, описанные на формальном языке, исключают неоднозначную трактовку свойств безопасности, при этом модель может быть верифицирована с использованием формальных методов, в частности, дедуктивной верификации. Та же модель может быть использована позже для тестирования реализации СЗИ.

В качестве языка моделирования может быть использован Event-B. Среда разработки Rodin [5] предоставляет для Event-B инструменты дедуктивной верификации, а язык обладает простым и понятным синтаксисом. Кроме того, имеется богатый опыт [6-8] использования Event-B для моделирования средств защиты информации операционных систем общего назначения. Но для создания тестового оракула и решения множества других задач необходимо воспроизвести поведение модели. Это непростая задача, учитывая, что модель написана в декларативной парадигме с прицелом на облегчение задачи дедуктивной верификации. Кроме того, для практического применения такой технологии необходимо избежать существенных ограничений на размер моделей и тестов, а значит, проверки должны выполняться максимально эффективно по времени и ресурсам.

Как демонстрирует предыдущая работа [9], из-за проблемы неопределенности этого нельзя добиться, не ограничивая стиль описания формальных моделей. В рамках работы был

сформулирован ряд предложений по ограничению синтаксиса моделей. Другим результатом работы было предположение, что наиболее многообещающим методом для эффективного воспроизведения поведения модели является генерация на основе модели программы на императивном языке.

Данное исследование посвящено разработке методики генерации компонентов такой программы на языке Python. Раздел 2 описывает контекст исследования, в нем перечисляются проблемы, связанные с воспроизведением поведения модели, и существующие подходы. Раздел 3 содержит постановку задачи, уточняя требования к разрабатываемому методу и обоснование выбора Python в качестве целевого языка. Раздел 4 описывает отображение различных конструкций Event-B в исходный код на Python. В Разделе 5 рассматривается практическая реализация предложенной методики.

2. Контекст исследования

2.1 Event-B

Язык Event-B можно условно разделить на две части: структурную часть [10], задающую порядок описания модели, и математический язык [11], позволяющий сформулировать необходимые утверждения при помощи формул.

Формулы математического языка Event-B основываются на формулах предикатов первого порядка и теории множеств. Все формулы должны быть согласованы с точки зрения используемых типов. В Event-B есть имеются следующие виды типов:

1. Базовые типы – логический (*TRUE* или *FALSE*) и целочисленный (произвольное целое число);
2. Пользовательский тип – задаваемый пользователем тип, отражающий некое абстрактное понятие. Объекты пользовательских типов могут быть связаны друг с другом только отношением равенства или отрицанием равенства;
3. Составные типы: упорядоченная пара и множество. Элементы упорядоченной пары могут иметь разные типы, множество состоит из элементов одного типа. Множество, состоящее из упорядоченных пар (обозначим левые элементы пар аргументами, а правые – значениями), задает отображение из множества аргументов во множество значений. Если аргументы при этом не повторяются, отображение будет являться функцией.

Со структурной точки зрения Event-B модель может состоять из компонентов двух видов: контекстов и машин.

Контекст позволяет определить предметную область моделируемой системы: в нем описываются используемые в модели константы и пользовательские типы. Типы представлены через несущие множества, состоящие из всех значений типа. Также контекст содержит аксиомы, определяющие значения и свойства констант и несущих множеств. Пример контекста, описывающего абстрактное понятие цвета (пользовательский тип *COLORS*) и две различные цветовые константы, приведен в Листинге 1.

Машина моделирует поведение системы. Состояние системы моделируется при помощи набора переменных. Набор допустимых состояний системы ограничивается при помощи инвариантов состояния. Переход между состояниями осуществляется при помощи событий, начальное состояние описывается событием инициализации.

Все события, кроме события инициализации, могут иметь входные параметры и охранные условия. Если охранные условия не выполняются для текущего состояния и полученного набора параметров, событие неприменимо, и переход по нему в новое состояние невозможен. Если охранные условия выполняются, то событие применимо, и можно осуществить переход, выполнив указанные в событии действия. Действия представляют собой параллельные

присваивания переменным новых значений. Новое значение может быть определено на основе значений констант, параметров и переменных в текущем состоянии (для события инициализации используются только константы: для него нельзя задать параметры, а текущего состояния с его переменными еще не существует).

Контекст может расширять другой контекст, добавляя новые несущие множества и константы и уточняя свойства определенных ранее. Машина может видеть контексты, а также уточнять другую машину. Уточняемая машина называется абстрактной, при этом переменные абстрактной машины могут входить в новую машину, а могут быть «забыты», а точнее, заменены новыми переменными. Событие новой машины может как расширять событие абстрактной машины, сохраняя все параметры и действия старого события, так и уточнять его, заменяя часть параметров и действий актуальными в новой машине.

Дедуктивная верификация Event-B модели подразумевает доказательство, что состояния, нарушающие инварианты, являются недостижимыми. Это так, если из выполнения охранных условий следует, что действия событий не нарушат инварианты (событие инициализации задает допустимое начальное состояние, все остальные события ведут из допустимых состояний в допустимые). Для проведения дедуктивной верификации может быть использована среда разработки Rodin и интегрированные в нее инструменты. Для примера из Листинга 1 единственный инвариант означает, что переменная *light* всегда будет сохранять свой тип (отметим, что модель допускает существование и использование бесконечного множества разных цветов). Доказательство будет тривиальным: для начального состояния выполнение инварианта *inv1* следует из аксиомы *axm1*, а для события *switch* – из охранного условия *grd1*.

```
context CTX

sets
    COLORS

constants
    red
    green

axioms
    @axm1: red ∈ COLORS
    @axm2: green ∈ COLORS
    @axm3: red ≠ green

end
```

```
machine MAC
sees CTX

variables
    light

invariants
    @inv1: light ∈ COLORS

events

    event INITIALISATION
    then
        @act1: light ≔ red
    end

    event switch
    any
        color
    where
        @grd1: color ∈ COLORS
        @grd2: color ≠ light
    then
        @act1: light ≔ color
    end

end
```

Листинг 1. Пример контекста и машины в Event-B.
Listing 1. Example of an Event-B context and a machine.

2.2 Связанные задачи

С точки зрения теории решаемая задача тесно связана с уже упомянутым понятием уточнения. Уточнение одной машиной другой является доказательным [12]: инварианты абстрактной машины выполняются и в уточненной. Тестируемая реализация также является

уточнением модели, но симуляционного характера [13-14]: в некотором смысле можно заменить одну другой и наблюдать одно и то же поведение. Выстраивая процесс симуляции самой модели, можно ориентироваться на оба подхода: с одной стороны, необходимо обеспечить корректное воспроизведение поведения, а с другой, облегчить возможность доказательства этой корректности. Этого можно добиться, если выполняемые в процессе симуляции действия будут простыми и легко проверяемыми.

Другой важной проблемой является проблема неопределенности [9]. Event-B модель описывает сразу все возможные сценарии поведения системы. Во время симуляции необходимо следовать одному из возможных сценариев, и модель может не содержать достаточно информации, чтобы сделать обоснованный выбор. Например, в Листинге 1 в событии инициализации можно написать другое действие: переменная *light* принимает любое из двух значений *red* и *green*. Тем не менее, можно утверждать, что подобные модели являются плохими спецификациями.

Также отметим, что для воспроизведения поведения модели на конкретном сценарии, этот сценарий необходимо где-то описать дополнительно. Примером сценария может быть трасса – линейная последовательность событий с параметрами. В общем случае сценарий может быть алгоритмом выбора последующего события, в котором параметры события могут быть неточными или вовсе не определенными.

Таким образом, на основе Event-B модели можно построить автоматический процесс проверки корректности выбранного сценария поведения. Для этого необходимо реализовать проверку охранных условий и выполнение действий всех событий. В случае произвольной модели или произвольного сценария построить такой процесс очень сложно, причем время выполнения процесса может сделать его неприменимым на практике. Но решение задачи для моделей и сценариев, лишенных неопределенности, позволит применить их для решения множества других задач.

Динамическая верификация. Метод динамической верификации [15] заключается в извлечении информации из работающей системы в виде трассы выполнения и последующей проверке этой трассы на основе спецификации поведения системы. В данном случае спецификацией является сама модель, а проверяемым сценарием является собранная трасса.

Тестирование на основе модели. Эта задача очень похожа на задачу динамической верификации по своей механике. В случае тестирования каждая полученная трасса будет являться результатом запуска теста.

Анимация модели. Анимацией модели называется процесс «отладки» модели, при котором сценарий составляется пользователем динамически. При этом пользователь получает информацию о состоянии модели. В самом простом случае аниматор анализирует составленную трассу, в более сложных он может предсказывать применимость каждого события и предлагать пользователю допустимые значения параметров (то есть те, для которых выполнены все охранные условия).

Проверка модели. Проверку модели можно назвать автоматической анимацией. Здесь сценарий будет алгоритмом, цель которого – достигнуть наибольшего количества различных по своим свойствам состояний, выбирая для этого применимые события с допустимыми параметрами. Целью проверки является поиск возможного пути в недопустимое состояние.

Все четыре задачи могут быть использованы в рамках жизненного цикла одной и той же модели. К примеру, можно представить себе следующую последовательность действий:

1. Разработчик модели редактирует модель и сразу же проверяет корректность вносимых изменений с помощью анимации, как наиболее простого способа проверки.
2. Разработчик модели после завершения очередного этапа запускает инструмент проверки модели, чтобы найти возможные ошибки, так как дедуктивная верификация слишком затратна для незаконченной модели.

3. Разработчик тестов пишет тесты для системы в соответствии с некоторой промежуточной версией модели. Это могут быть проверки некоторых типичных для подобных систем сценариев (например, создание файла в файловой системе ОС).
4. Разработчик системы реализует некоторый функционал системы. В таком случае можно провести тестирование, которое способно выявить ошибки как в системе, так и в разрабатываемой модели.
5. Финальная версия модели верифицируется, например, при помощи дедуктивной верификации.
6. Разрабатываемая система верифицируется на основе финальной версии модели при помощи динамической верификации.

2.3 Существующие методы воспроизведения поведения модели

Наиболее естественным образом поведение модели можно воспроизвести, используя средства логического программирования. Такой подход позволяет искать решение для произвольной модели и произвольного сценария. Он используется в наборе инструментов ProB [16], включающем в себя инструменты анимации и проверки моделей. Приоритетом разработчиков ProB является поддержка всех возможностей языка, а не эффективность. Однако на практике такой подход очень требователен к ресурсам, что порождает множество ограничений. ProB не позволяет обрабатывать большие наборы значений, переменных состояния и событий в сценариях, что делает его неприменимым для сложных систем. Кроме того, данный метод не является прозрачным: ProB использует проприетарный компилятор языка SICStus Prolog, что мешает как диагностике, так и потенциальной доработке.

Еще один возможный подход заключается в интерпретации формул Event-B программой на императивном языке. Такой подход использовался в аниматорах Brama [17] и AnimB [18], однако они решали те же задачи, что и ProB, несколько уступая последнему в возможностях. Данные инструменты не обновлялись более десяти лет. В рамках предыдущего исследования [9] был создан прототип интерпретатора Event-B, выполняющий задачу для подмножества моделей с детерминированным поведением эффективно по времени. Но это решение не является лучшим: интерпретация формул менее эффективна, чем выполнение единожды сгенерированной по тем же правилам программы, а пользователю все еще приходится доверять создателю интерпретатора до тех пор, пока не обнаружится несоответствие между наблюдаемым поведением и ожидаемым.

Таким образом, был выбран подход с генерацией кода на основе модели, который является не только наиболее эффективным по времени, но и наиболее прозрачным. Пользователь получает доступ ко всему исполняемому во время проверки коду и может его анализировать, не рассматривая инструмент трансляции. Кроме того, к существующим инструментам для работы с Event-B моделями автоматически добавляются все существующие инструменты для целевого языка программирования.

Существует множество [19] инструментов генерации кода на основе Event-B, однако ни один из них не поддерживает конструкции языка в достаточной мере. Например, B2C [20] не только не поддерживает контексты (а следовательно, константы и пользовательские типы), но и концентрируется на генерации сценария, не позволяя выбирать произвольный порядок событий. Наиболее полным и подходящим для наших задач является EventB2Java [21]. Данный инструмент использовался в некоторых найденных в открытом доступе проектах для создания оракула unit-тестов реализации. Однако последняя версия EventB2Java не поддерживает трансляцию кванторов, на которых строится большинство инвариантов безопасности. Кроме того, генерируемый код не рассчитан на чтение: формулы существенно меняют свой вид, что обусловлено выбором языка, а все охранные условия события записываются в одну общую строку.

В ходе предыдущих исследований [6] была предложена методика трансляции Event-B моделей в код на Python, которая использовалась для написания исполняемых версий в модели вручную. Однако методика не описывала автоматизированного процесса генерации кода. Попытка построить такой процесс на основе предложенных в методике правил выявила множество проблем, связанных с типами данных и несоблюдением исходной семантики Event-B.

3. Постановка задачи

Целью данного исследования является исследование методики воспроизведения поведения модели при помощи генерации кода. Перед постановкой задачи необходимо ограничить область исследования и сформулировать ряд требований к генерируемой программе.

Во-первых, в некоторых случаях задача генерации кода может подразумевать генерацию безопасной реализации по модели (например, для моделей алгоритмов). Мы не ставим перед собой такой задачи, так как рассматриваемые нами модели слишком абстрактны и отражают лишь некоторые свойства системы.

Во-вторых, мы не пытаемся сгенерировать сценарий для проверки. Пользователь сам задает необходимый сценарий, используя сгенерированные функции. Наша задача ограничивается генерацией библиотеки для воспроизведения поведения модели.

В-третьих, частью задачи воспроизведения поведения может являться автоматический подбор значений модельных констант на основе текста модели. Эта задача была подробно рассмотрена в предыдущей работе [9], в рамках данного исследования мы предполагаем, что значения констант задаются пользователем. Интеграция автоматического подбора констант с разрабатываемым методом является техническим вопросом.

Процесс генерации кода, принимающий на вход Event-B модель и генерирующий на ее основе библиотеку на выбранном целевом языке программирования (ЯП), мы будем называть трансляцией, а выполняемую этим процессом программу – транслятором.

Полученная в результате трансляции библиотека может использоваться в программе пользователя, задающей проверяемый сценарий. Примером такой программы может быть интерпретатор собранных трасс. Общая схема процесса воспроизведения поведения модели представлена на рис. 1.

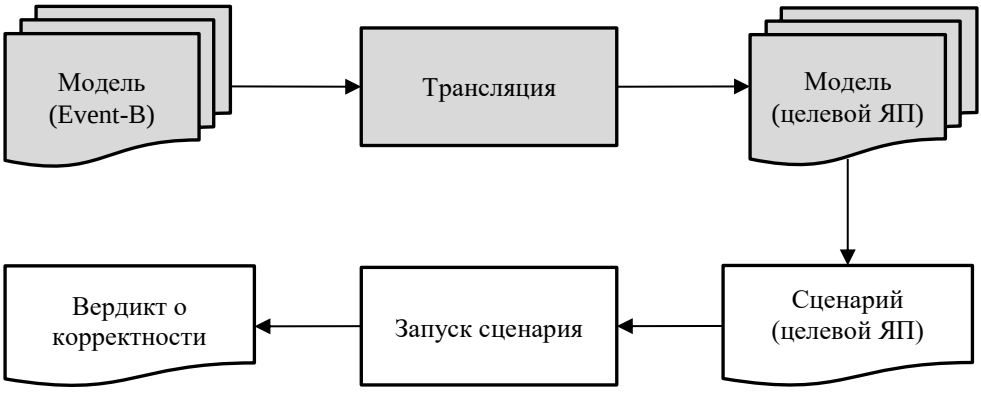

Рис. 1. Общая схема воспроизведения поведения модели.
Fig. 1. Reproducing the behavior of a model.

Выдвинем требования к транслятору и получающейся в результате трансляции библиотеке.

Мощность метода. Транслятор должен поддерживать максимально возможное количество конструкций Event-B. Поддержка произвольных моделей невозможна, поэтому необходимо наложить на модель некоторые разумные ограничения, связанные с неопределенностью. Также вполне допустимо иногда требовать от разработчика модели выбрать один способ записи из множества возможных. С другой стороны, разработчик модели должен иметь возможность использовать как можно больше конструкций.

Эффективность по времени. Программа должна выполняться эффективно по времени, в частности, эффективно вычислять кванторы.

Прозрачность действий. Для разработчика модели генерируемый исходный код должен легко соотноситься с текстом модели, чтобы он мог убедиться в корректности трансляции. Для разработчика тестов генерируемый исходный код должен выглядеть простым и понятным, так как он будет более знаком с целевым ЯП, чем с Event-B. Также желательно, чтобы разработчику тестов не пришлось изучать целевой ЯП только ради этого.

Использование библиотечных функций и классов допустимо для достижения близости текста генерируемого кода к тексту модели, а также в случаях, когда обойтись без этого невозможно. Такие функции и классы не должны скрывать от пользователя сложное поведение или обладать неочевидными побочными эффектами.

Исходя из сформулированных требований, мы выдвигаем следующие требования к целевому ЯП:

1. В языке должен быть соблюден баланс между скоростью выполнения и читаемостью кода;
2. Язык должен быть у пользователя в активе и применяться в рамках того же проекта, например, для тестирования;
3. Язык должен иметь удобные структуры для представления типов Event-B: пользовательских типов, упорядоченных пар, множеств;
4. Язык должен предоставлять возможность для преодоления трудностей, которые возникают на стыке математики и программирования: переполнение целочисленных типов, возможность легкой реализации больших и потенциально бесконечных множеств, возможность записи кванторов и т. д.

С учетом сформулированных требований в качестве целевого ЯП хорошо подходит Python: он широко распространен и обладает всеми необходимыми свойствами.

Задачей исследования является разработка метода генерации программного кода на Python на основе Event-B модели, удовлетворяющего перечисленным выше требованиям.

4. Решение

4.1 Типы данных

Для трансляции формул Event-B в выражения Python необходимо построить отображение между типами в обоих языках (см. табл. 1).

Базовые типы. Встроенными базовыми типами в Event-B являются два типа: целочисленный и логический. Эти базовые типы существуют и в языке Python. В Python целые числа обладают представлением, не ограниченным размерностью, в отличие, например, от языков C или Java. Кроме того, Python поддерживает все необходимые операции над этими типами данных, поэтому никаких дополнительных действий или ограничений не требуется.

Пользовательские типы. Значение объекта пользовательского типа складывается из ссылки на несущее множества и уникального числа. Например, для цветовой константы возможно значение *COLORS2*, а все объекты этого типа будут экземплярами класса *ColorsItem*. Для

удобства отладки при создании нового экземпляра можно указать псевдоним: имя константы или строковое представление реальных данных.

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

Тип	Event-B	Python
Логический	$BOOL$	$bool$
Целочисленный	$\mathbb{Z}$	int
Пользовательский	A	$AItems$
Несущее множество	$\mathbb{P}(A)$	$CarrierSet[A]$
Множество	$\mathbb{P}(A)$	$frozenset[A]$
Упорядоченная пара	$A \times B$	$tuple[A, B]$

Объекты пользовательского типа должны поддерживать операции проверки равенства и неравенства. Это легко сделать благодаря числовому значению в каждом из объектов.

Более сложной задачей является отображение несущих множеств типов в Python. С точки зрения Event-B, несущие множества являются обычными множествами и могут быть использованы в различных формулах. При этом несущие множества можно разделить на три вида:

- Перечислимые – для такого несущего множества известен его размер, причем каждый его элемент является константой в рамках модели. Примером такого множества может быть множество возможных прав в файловой системе. Для них удобно представление в виде обычного множества.
- Бесконечные – примером потенциально бесконечного несущего множества может быть множество строк произвольной длины, которое порождает огромное количество возможных комбинаций даже для коротких строк. Их представление в виде обычного множества будет некорректным, поэтому они должны быть представлены в виде типа данных.
- Конечные – для такого множества известен его размер, но элементов слишком много, чтобы описывать каждый заранее в виде константы. Примером может быть множество файлов в файловой системе. Такое множество фактически не имеет смысла заполнять, пока элементы не фигурируют в состоянии модели: оно может наполняться динамически, пока множество не достигнет своего лимита. Но пока множество не заполнено, представление в виде обычного множества также будет некорректным.

Для того, чтобы сделать возможным существование бесконечных и незаполненных конечных множеств, сохранив при этом возможность использование несущих множеств в различных формулах, необходимо объединить представление в виде типа и множества значений в одном объекте. Обойтись стандартными средствами Python в данном случае невозможно.

Для каждого из видов несущих множеств в библиотеку была добавлена функция, возвращающая экземпляр класса *CarrierSet*. Бесконечные множества поддерживают операции проверки типа элемента и подмножеств несущего множества. Конечные множества поддерживают операцию взятия мощности множества. Перечислимые множества поддерживают любые операции над множествами.

Также все виды несущих множеств поддерживают операцию разности с конечным множеством, так как эта операция распространена в Event-B моделях.

Составные типы. В Event-B существует два вида составных типов: упорядоченная пара и множество. Упорядоченная пара может быть легко отображена в кортеж из двух элементов.

К сожалению, множества не могут быть отображены в множество Python: стандартные множества в Python являются изменяемыми, что не позволяет построить множество множеств. Для представления множеств используется неизменяемый тип *frozenset*.

В Event-B множество упорядоченных пар является отображением, а в случае, когда первые элементы пар не повторяются – функцией. В монографии [6] для них предлагается представление в виде словаря. Это позволит эффективно выполнять такие операции, как взятие значения функции или множества значений отображения по множеству аргументов, вычитание области определения, взятие области определения. Однако использование словарей порождает множество проблем. В частности, отображения могут участвовать во всех операциях, типичных для множеств: объединение, вычитание множества элементов и т. д. Для словарей эти операции не реализованы.

Так как данная оптимизация не может быть реализована при помощи стандартных средств Python и не является принципиальной, было решено от нее отказаться. Отображения и множества представлены в виде множеств пар, а все операции над отображениями реализованы при помощи простейших библиотечных функций.

4.2 Формулы

Также необходимо транслировать сами формулы Event-B: предикаты и выражения (см. табл. 2). Формулы предварительно обрабатываются синтаксическим анализатором, поэтому при трансляции мы работаем с синтаксическим деревом.

Импликация. В Python нет логической операции для импликации. Чтобы сохранить ленивую семантику вычислений, импликация выражается через другие логические операции, а не через новую библиотечную функцию.

Предикат раздела. Также в Event-B существует предикат раздела $partition(A, B, C)$. Этот предикат эквивалентен выражению $A = B \cup C \wedge B \cap C = \emptyset$. Можно записать его именно в таком виде, однако $partition$ может принимать на вход и большее число аргументов, соответственно, длина эквивалентного выражения будет увеличиваться. В данном случае будет удобнее написать библиотечную функцию.

Кванторы. Трансляция произвольного квантора в выражение на языке программирования является сложной задачей. Это приводит к тому, что некоторые инструменты не транслируют кванторы, например, так поступает EventB2Java. Однако трансляция необходима, так как кванторы часто используются в охранных условиях для записи свойств сложных структур данных.

В Python для записи кванторов можно использовать функции *all* и *any*. Тогда вычислить квантор можно перебором: функции способны работать на основе генераторов значений, сохраняя семантику ленивых вычислений. Если среди возможных значений подкванторных переменных найден пример для *any* или контрпример для *all*, перебор немедленно прекращается и возвращается результат.

Однако решить задачу перебором невозможно, если область значений переменных бесконечна. Для эффективного вычисления кванторов предлагается ограничить область значений для каждой подкванторной переменной эффективно перебираемым множеством. Такое же требование выдвигается, например, для аниматора AnimB. Как правило, такую область значений легко выделить из формулы.

В табл. 2 приведены лишь два частных случая правил трансляции кванторов. В них присутствует только одна переменная x, область значений которой ограничена множеством X. Отметим, что для квантора всеобщности ограничение области значения записывается при помощи импликации, а для квантора существования – с помощью конъюнкции. В краткой

сводке языка Event-B [22] кванторы записаны именно в таком виде. Мы требуем придерживаться такой же формы записи.

Табл. 2. Некоторые правила трансляции для предикатов.
Table 2. Some of predicate translation rules.

Event-B	Python	Event-B	Python
$\top$	`True`	$E \in S$	`E in S`
$\bot$	`False`	$E \notin S$	`E not in S`
$P \wedge Q$	`P and Q`	$E \subseteq S$	`E <= S`
$P \vee Q$	`P or Q`	$E \not\subseteq F$	`not (E <= S)`
$P \Rightarrow Q$	`not P or Q`	$E \subset F$	`E < S`
$P \Leftrightarrow Q$	`P == Q`	$E \not\subset F$	`not (E < S)`
$\neg P$	`not P`	$finite(S)$	`finite(S)`
$\exists x \cdot x \in X \wedge P$	`any (True for x in X if P)`	$m > n$	`m > n`
$\forall x \cdot x \in X \Rightarrow P$	`not any (True for x in X if not P)`	$m < n$	`m < n`
$E = F$	`E == F`	$m \geq n$	`m >= n`
$E \neq F$	`E != F`	$m \leq n$	`m <= n`

Для квантора существования все значения, на которых формула ложна, отфильтровываются при помощи ключевого слова *if*. Это необходимо, чтобы в сгенерированном коде сохранялся порядок частей формул. Рассмотрим формулу 1, моделирующую утверждение «в системе существует хотя бы один пользователь-администратор», и два альтернативных способа трансляции этой формулы в Python.

$$\exists u \cdot u \in Users \wedge isAdmin(u) = TRUE \quad (1)$$

```
any(isAdmin(u) == True for u in Users)
```
(2)

```
any(True for u in Users if isAdmin(u) == True)
```
(3)

Даже в таком простом случае визуально легче сопоставить выражения 1 и 3, при этом чем длиннее формула, тем более важным становится этот аспект.

Для функции *all* эта проблема также актуальна, но мы не можем воспользоваться фильтром, иначе мы просто потеряем все контрпримеры. Поэтому в табл. 2 квантор всеобщности превращается в отрицание квантора существования, при этом сохраняется порядок формул:

$$\forall x \cdot \big(P(x) \Rightarrow Q(x)\big) \Leftrightarrow \neg \exists x \cdot (P(x) \wedge \neg Q(x)) \quad (4)$$

Алгоритм эффективной трансляции квантора (потенциально с несколькими подкванторными переменными, области значения которых могут зависеть друг от друга) несложно построить, если для формулы, задающей область значений подкванторных переменных, выполняются следующие условия:

- Формула является конъюнкцией, разбор которой ведется слева направо;
- Если в очередном конъюнкте переменная (или нескольких переменных) упоминается впервые, то конъюнкт должен задавать область значения этой переменной (или нескольких переменных);
- Конъюнкты, задающие область значения переменных, наглядно транслируются в выражение *for…in…if…*: например, конъюнкт $x \mapsto a \in f$, содержащий одну переменную x, можно транслировать в выражение *for (x, val) in f if val == a*.

Для большинства выражений в Python существуют аналоги: это, например, арифметические операции и операции над множествами. Выражения, для которых в Python нет стандартных аналогов, можно определить в виде библиотечных функций, реализация которых, как правило, может быть записана в одну строку. Тем не менее, стоит подробнее остановиться на некоторых из них (табл. 3).

Табл. 3. Некоторые правила трансляции для выражений.
Table 3. Some of expression translation rules.

Event-B	Python	Event-B	Python
$\emptyset$	`frozenset()`	$m..n$	`range(m, n+1)`
$\{E\}$	`frozenset((E,))`	$dom(R)$	`relation_domain(R)`
$\{E, F\}$	`frozenset((E,F))`	$ran(R)$	`relation_range(R)`
$\{E \mid x \in X\}$	`frozenset(E for x in X)`	$S \lhd R$	`restrict_domain(R, S)`
$\mathbb{P}(S)$	`powerset(S)`	$S \lhd\!\!\!- R$	`subtract_domain(R, S)`
$E \cup F$	`E | F`	$S \rhd R$	`restrict_range(R, S)`
$E \cap F$	`E & F`	$S \rhd\!\!\!- R$	`subtract_range(R, S)`
$E \setminus F$	`E - F`	$S \leftarrow R$	`relation_override(R, S)`
$E \mapsto F$	`(E, F)`	$R[S]$	`relation_image(R, S)`
$E \times F$	`cartesian_product(E, F)`	$f(E)$	`function_value(f, E)`
$E = F$	`E == F`	$E \leftrightarrow F$	`relations(E, F)`
$card(S)$	`len(S)`	$E \nrightarrow F$	`partial_functions(E, F)`
$\mathbb{Z}$	`INT`	$E \rightarrow F$	`total_functions(E, F)`

Определение множества через предикат. Множества могут определяться через выражение вида $\{F \mid P\}$, где F – формула, по которой вычисляются элементы множества, а P – предикат, содержащий переменные. В Python тоже присутствует такая форма записи, причем именно ей мы пользовались выше при трансляции кванторов. При этом для ее реализации над предикатом P достаточно выполнить ту же процедуру, что и при трансляции квантора.

Так, например, формула 5 транслируется в выражение 6.

$$\{x + y \mid x \in X \wedge y \in Y \wedge x \neq y\} \quad (5)$$

```
frozenset(x + y for x in X for y in Y if x != y)
```
(6)

Множества отображений, функций, подмножеств. Некоторые операции над множествами подразумевают построение новых множеств из комбинаций их элементов. В случае конечных множеств это приводит к проблеме комбинаторного взрыва числа элементов. Нередко эти операции применяются к бесконечным множествам, так как через них может выражаться тип объекта в Event-B.

Так, предикат 7 означает, что *messages* является множеством строк, но в записи фигурирует множество всех подмножеств бесконечного множества всех возможных строк. Предикат 8 означает, что *links* является частично определенной функцией из множества строк в множество файлов, при этом фигурирует множество всех таких функций, которые только возможны.

$$messages \in \mathbb{P}(STRINGS) \quad (7)$$

$$links \in STRINGS \nrightarrow FILES \quad (8)$$

Разумеется, мы не можем представлять такие объекты в виде обычных множеств. Поэтому для таких операций мы вынуждены реализовать новые классы объектов, содержащих в себе только множества, на основе которых они строятся. Если исходное множество бесконечно, то пользователь должен использовать результат операции только для определения типа. Если же исходное множество является конечным, то из такого объекта при необходимости можно получить все возможные элементы при помощи итератора.

4.3 Присваивания

Присваивания значений переменным в Event-B можно разделить на определенные («:=», присваивание значения) и неопределенные («: ∈», выбор значения из множества; «: |», выбор значения, удовлетворяющего предикату). В статье [9] мы сделали вывод, что в пригодных для тестирования моделях необходимо отказаться от выбора значения из множества, но при этом возможен выбор значения по предикату, если он выражает конструкцию if-then-else. В выражении 9 переменная *var* принимает значение *A*, если условие *P* выполняется, и значение *B* в противном случае.

$$var \coloneqq P \wedge var' = A \vee \neg P \wedge var' = B \tag{9}$$

Еще одним важным отличием Event-B от императивных языков программирование является параллельное присваивание переменных в рамках одного события.

Так, в присваиваниях 10 и 11 значения переменных a и b меняются местами. Если транслировать эти присваивания на Python напрямую, то мы получим расхождение с моделью: обе переменные будут содержать значение переменной b. Этот момент учитывается во многих инструментах (например, в EventB2Java), но в других инструментах может быть упущен: например, этот момент никак не отражен в кратком описании методики трансляции на Python в рамках монографии [6].

$$a \coloneqq b \tag{10}$$

$$b \coloneqq a \tag{11}$$

Отметим, что присваивания Event-B все же могут транслироваться в присваивания Python в рамках события инициализации. Как упоминалось ранее, при описании события инициализации нельзя ссылаться на предыдущие значения переменных, так как состояние машины еще не построено.

Прямое присваивание. При трансляции обычных присваиваний вида $x := A$ необходимо выделить имя переменной *x* и транслировать выражение *A* для получения ее значения.

Присваивание в точке. Для функций синтаксис Event-B позволяет записать присваивание вида $f(x) := A$, означающее замену значения функции *f(x)*. Остальные значения останутся прежними. Этот случай можно свести к предыдущему при помощи оператора перегрузки, получив присваивание $f := \{x \mapsto A\} \leftarrow f$. Такой же подход используется в Rodin.

Условное присваивание. Для трансляции присваивания в формуле 9 используется аналог тернарной операции в Python. Новое значение для переменной *var* будет вычисляться как *A if P else B*. Здесь структура выражения отличается от принятой в Event-B, но такой вид будет более привычен для разработчика Python и не должен вызвать затруднений у разработчика модели, так как такая запись проще, чем используемая в Event-B.

4.4 Компоненты модели

Модель. При трансляции контекстов и машин возможны два подхода: первый – трансляция каждого компонента в отдельный модуль и второй – создание одной «плоской» модели, содержащей в себе информацию из разных компонентов. Выбор подхода зависит от того, необходимо ли отдельно воспроизводить поведение абстрактных машин: в таком случае полезно, чтобы они являлись отдельными модулями.

Для воспроизведения поведения модели в рамках задач тестирования реализации достаточно воспроизвести поведение только машины самого верхнего уровня. Если модель прошла дедуктивную верификацию, то корректность работы абстрактной машины следует из корректности работы машины верхнего уровня. Кроме того, в абстрактных машинах может быть оказаться недостаточно информации для воспроизведения ее поведения. По этим причинам на данном этапе мы поддерживаем уточнение машины только в контексте расширения поведения абстрактной машины (либо в принципе запрещая «забывать» переменные абстрактной машины, либо игнорируя формулы, содержащие «забытые» переменные).

Выбранная пользователем машина транслируется в одну «плоскую» модель, содержащую в себе объекты контекстов и абстрактных машин. Этот же подход используется, например, в EventB2Java.

Отметим, что такой подход окажется слишком ограничен, если рассматривать задачу проверки модели, так как в этом случае модель не будет верифицирована. Поддержка более сложных вариантов уточнения абстрактных машин с воспроизведением их поведения является областью дальнейших исследований.

Контекст. Элементы видимого машиной контекста становятся атрибутами класса *Machine*.

Несущие множества транслируются в два атрибута класса *Machine*: класс объектов и множество таких объектов. Так, несущее множество *STRINGS* транслируется в класс *StringsItem* и во множество всех объектов этого класса *STRINGS = infinite_carrier_set('STRINGS', StringsItem)*. По умолчанию все несущие множества считаются бесконечными, если это не так, они должны быть переопределены пользователем или автоматическим процессом в классе-наследнике класса *Machine*. Необходимость переопределения обуславливает определение элементов контекстов как атрибутов класса, а не атрибутов модуля (глобальные переменные и типы), что кажется более естественным. Создание класса для несущего множества не является необходимым для выполнения программы на Python, но оно необходимо для аннотирования параметров функций типами значений. Это позволяет быстрее находить ошибки использования соответствующих параметров при статическом анализе перед выполнением программы.

Константа транслируется в атрибут класса *Machine* с соответствующим именем. Если константа имеет тип, определенный пользователем, ей присваивается новый экземпляр соответствующего класса *Item*, при этом имя константы передается в качестве псевдонима для нового объекта. Если константа является множеством, логической константой или числом, ей присваивается значение по умолчанию. Значения этих констант должны быть переопределены в классе-наследнике класса *Machine*.

Так как контекст не меняется во время воспроизведения поведения модели, аксиомы могут проверяться единожды, например, в начальном состоянии машины. Каждая аксиома транслируется в тест для состояния машины, проверяющий, что значение аксиомы истинно. Этот тест означает, что пользователь корректно переопределил значения констант и несущие множества. Тесты содержатся в отдельном от машины модуле.

Машина. Переменные транслируемой машины становятся атрибутами класса *Machine*. Инварианты машин становятся еще одним модулем тестов для начального состояния машины. Поскольку для воспроизведения будет использоваться не класс *Machine*, а его класс-наследник, то инварианты надо проверять для каждого состояния каждого объекта класса-наследника. Однако такие проверки удается выразить в виде отдельных тестов только для начального состояния. Для всех остальных состояний необходимо проверять инварианты для каждого нового состояния при воспроизведении сценариев.

События. Событие *INITIALISATION* транслируется в метод *__init__* класса *Machine*. Все действия этого события транслируются в присваивания, так как из-за отсутствия текущего состояния все присваивания события инициализации независимы.

Остальные события транслируются в функции, которые должны вычислить значения всех охранных условий соответствующего события и подготовить для выполнения действия события.

Охранные условия зависят от значений параметров и текущего состояния машины. При их фиксации охранное условие получит одно из трех возможных значений: охранное условие истинно, охранное условие ложно, охранное условие невычислимо (при вычислении приводит к исключительной ситуации). У события может быть несколько охранных условий, и для их проверки нужно выбрать порядок проверки. Наиболее логичным было бы проверять охранные условия в порядке их следования в модели. Однако этот порядок может приводить к тому, что проверка охранных условий завершится исключительной ситуацией, что означает ошибку в тесте, требующей исправления. На самом же деле ошибки может не быть, потому что одно из последующих охранных условий будет ложно Это означает, что общее охранное условие события, являющееся конъюнкцией отдельных охранных условий, тоже ложно. Такая ситуация может случиться по следующей причине.

При тестировании выполнение некоторой операции может быть запрошено, даже если событие, ей соответствующее, неприменимо к текущему состоянию модели (например, попытка открыть файл, к которому нет доступа, чтобы проверить, что попытка действительно завершится отказом). При этом такая операция в системе должна завершиться неуспешно. Обратное также верно: если операция в системе завершилась неуспешно, в модели должно быть ложно хотя бы одно из охранных условий.

Параметры событий в Event-B моделируют как входные данные, так и результаты выполнения операции (побочный эффект операции). Неуспешная операция не должна приводить к побочным эффектам в системе. Соответственно, при воспроизведении неприменимого события для параметров, означающих результат операции, невозможно однозначно подобрать значение. Попытки читать значения таких параметров должны классифицироваться как ошибка (исключительная ситуация при выполнении охранного условия). Однако если после такого охранного условия идет ложное охранное условие, то такая ошибка закономерна (файл не создан – нет группы созданного файла, так как нет прав для создания файла), тест не должен сигнализировать об этой ошибке.

Можно избежать возникновения ошибок из-за отсутствующих данных, если при вычислении значений охранных условий сохранять возникающие исключительные ситуации, но не завершать тест. Назовем это результатом проверки охранных условий. Этот же механизм можно использовать для реализации параллельных присваиваний: сначала вычислить все результаты (сохраняя возникшие исключительные ситуации) и только потом применить их к состоянию модели.

Альтернативным решением было бы дать пользователю задать вручную порядок проверки охранных условий, но такой подход усложняет работу, не давая взамен никаких преимуществ.

События транслируются в функции, принимающие на вход экземпляр класса *Machine* и набор параметров. Все параметры являются опциональными на тот случай, если событие неприменимо и значение параметра невозможно однозначно подобрать. Функции возвращают объект, содержащий результаты проверки охранных условий и выполнения действий.

В Листинге 2 продемонстрированы фрагменты результата трансляции для примера из Листинга 1. Перегруженный оператор инверсии ~ используется для получения либо значений параметров, охранных условий и выражений в присваиваниях, либо соответствующей ошибки, если значение получить невозможно.

После получения результата функции-события необходимо проверить охранные условия. Если ожидается, что событие неприменимо, среди охранных условий должно быть нарушенное (лямбда-функция охранного условия вернула *False*). Если событие

неприменимо, но такое условие не найдено, то либо модель неполна, либо пользователь допустил ошибку в процессе воспроизведения, не передав необходимый параметр. Если все охранные условия выполняются (лямбда-функции вернули *True*), событие применимо и состояние системы обновляется при помощи вычисленных результатов присваиваний.

```
class Machine:
    class ColorsItem: pass
    COLORS = infinite_carrier_set('COLORS', ColorsItem)
    red = constant('red', ColorsItem)
    green = constant('green', ColorsItem)
    def __init__(self):
        self.light = self.red

def switch(m: Machine,
           _color: Optional[Machine.ColorsItem]) -> Event:
    color = Parameter(_color)
    _grd1 = Guard(grd1, lambda: ((~color) in m.COLORS))
    _grd2 = Guard(grd2, lambda: ((~color) != m.light))
    _act1 = Action(act1, m, 'light', lambda: ((~color)))
    return Event(event, _grd1, _grd2, _act1)
```

Листинг 2. Результат трансляции машины и события для модели из Листинга 1.
Listing 2. Translation result for the machine and the event for the model from Listing 1.

Функционал проверки применимости события и выполнения его действий в случае применимости реализован в библиотечных функциях *assertTrue* и *assertFalse*, принимающих на вход экземпляр класса *Event*, который вернула функция события.

5. Практическая реализация

В рамках инструмента АНИС [23] на основе предложенной в статье методики был реализован функционал трансляции формальной модели. Помимо самой трансляции, инструмент также анализирует модель, подсказывая пользователю необходимые для успешной трансляции исправления. Как правило, исправления связаны с необходимостью добавить область значений подкванторных переменных.

Также были расширены возможности отладки. Вид, представленный в Листинге 2, неудобен для отладки из-за лямбда-функций. Кроме того, в ходе выполнения теста становятся известны результаты проверок охранных условий, что позволяет упростить семантику выполняемых при отладке действий. Для каждого вызова функции события может быть сгенерирована ее упрощенная копия, в которой порядок охранных условий будет следовать правилу: сначала идут выполненные условия, затем нарушенные, затем те, вычисление которых завершается с ошибкой. Предположим, что мы дважды вызвали событие *switch(red)* из Листинга 2. Для второго вызова может быть сгенерирована функция из Листинга 3, причем нарушенный *grd2* будет последним согласно правилу, а не из-за порядка в модели.

Также необходимо оценить эффективность работы инструмента по времени. Для этого был проведен повторный эксперимент, используемый в предыдущем исследовании [9]. Эксперимент проводился на модели контроля доступа ОС Linux, инструментах, поддерживающих эту модель и наборе из 39000 тестовых трасс. Результаты эксперимента приведены в табл. 4.

Код, сгенерированный АНИС, все же уступает в эффективности коду, написанному вручную. Причина отставания заключается в том, что написанная вручную модель лучше справляется с кванторами: при ее написании использовались те же оптимизации, что и в интерпретаторе [9], реализованном на Java, в то время как АНИС транслирует формулу максимально близко к оригиналу. Кроме того, в написанной вручную модели функции были представлены в виде

словарей, а значит, некоторые операции над функциями выполнялись быстрее. Это подтвердили данные, полученные при профилировании инструмента: самой затратной по времени операцией оказалась *subtract_domain*.

В перспективе можно добавить режим трансляции с оптимизацией, который позволит преобразовывать формулы к более эффективному для вычислений виду. Но даже без оптимизаций было достигнуто значительное ускорение относительно реализованного ранее интерпретатора. Одним из дальнейший направлений деятельности является проверка, достаточна ли такая скорость для онлайн-мониторинга системы в рамках динамической верификации.

```
def switch_2(m: Machine,
             _color: Optional[Machine.ColorsItem]) -> Event:
    color = assert_is_not_none(_color)

    #grd1
    assert (color in m.COLORS)
    #grd2
    assert (color != m.light)

    #act1
    _light = color

    m.light = _light
```

Листинг 3. Отладочная версия события.
Listing 3. Debug version of the event.

Табл. 4. Сравнение времени анализа трасс для различных инструментов.
Table 4. Comparison of trace analysis times for different tools.

Python, вручную	**Python, АНИС**	**Интерпретатор (Java)**	**ProB**
1 мин.	4 мин.	8 мин.	> 10 час.

6. Заключение

В ходе исследования были предложены требования к процессу трансляции формальных моделей в исполняемый код с учетом возможности широкого промышленного применения: распространенность целевого языка программирования, легкость сопоставления частей генерируемого кода с текстом исходной модели, а также простота и читаемость генерируемого кода с точки зрения программиста.

С учетом этих критериев, для моделей, описывающих системы с детерминированным поведением, была описана методика трансляции в код на языке программирования Python. Данная методика обладает достаточной мощностью для ее применения на реальных моделях контроля доступа ОС общего назначения.

Предложенная методика была успешно реализована в рамках инструментария АНИС. Реализация подтвердила гипотезу, что подход с автоматической генерацией кода более эффективен по времени, чем реализованный ранее [9] подход с интерпретацией формул. При этом инструментарий АНИС позволяет не только воспроизводить поведение модели, но и предоставляет пользователю не реализованные ранее возможности анализа модели и отладки процесса воспроизведения.

Разработанный инструмент трансляции может применяться при создании инструментов анимации моделей, проверки моделей, тестирования на основе моделей и динамической верификации реализации, что является направлением дальнейших исследований.

Список литературы / References

[1]. 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).

[2]. 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).

[3]. ГОСТ Р 59453.3. Защита информации. Формальная модель управления доступом. Часть 3. Рекомендации по разработке. – Введен 2025-03-31 – Москва: Стандартинформ, 2025. / State Std. GOST R 59453.3. Information protection. Recommendations on verification of formal access control model. Part 3. Recommendations on development. Moscow, 2025 (in Russian).

[4]. ГОСТ Р 59453.4. Защита информации. Формальная модель управления доступом. Часть 4. Рекомендации по верификации средства защиты информации, реализующего политики управления доступом, на основе формализованных описаний модели управления доступом. – Введен 2025-03-31 – Москва: Стандартинформ, 2025. / State Std. GOST R 59453.4. Information protection. Recommendations on verification of formal access control model. Part 4. Recommendations for verification of information security features that implement access control policies based on formal descriptions of the access control model. Moscow, 2025 (in Russian).

[5]. 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.

[6]. 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).

[7]. 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.

[8]. 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.

[9]. Карнов А.А. Проблема неопределенности в анализе трасс на основе высокоуровневых моделей в контексте динамической верификации. Труды ИСП РАН, 2024, том 36, вып. 4, стр. 169–182. DOI: 10.15514/ISPRAS-2024-36(4)-13 / Karnov A.A. Uncertainty Problem in High-Level Model-Based Trace Analysis as Part of Runtime Verification. Proceedings of the ISP RAS, 2024, vol. 36, no. 4, pp. 169–182. (In Russian). DOI: 10.15514/ISPRAS-2024-36(4)-13

[10]. J.-R. Abrial, The Event-B Modelling Notation, Available at: https://wiki.event-b.org/index.php/Event-B_Modelling_Language

[11]. Metayer C., Voisin L., The Event-B Mathematical Language, Available at: https://wiki.event-b.org/index.php/Event-B_Mathematical_Language

[12]. Abrial J.-R., Hallerstede S., Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. In Fundamenta Informaticae, 2007, vol. 77, pp. 1–28. DOI: 10.5555/2366448.2366450.

[13]. Börger E., ASM Refinement Method. In Formal Aspects of Computing, vol. 15, pp. 237–257, Springer-Verlag, 2003. DOI: 10.1007/s00165-003-0012-7
[14]. Efremov D., Shchepetkov I., Runtime verification of Linux kernel security module. In International Symposium on Formal Methods, 2019, pp. 185–199. Available at: https://arxiv.org/pdf/2001.01442
[15]. 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.
[16]. 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.
[17]. Servat, T., BRAMA: A New Graphic Animation Tool for B Models. In Julliand, J., Kouchnarenko, O. (edsitors) Formal Specification and Development in B. Lecture Notes in Computer Science, vol. 4355. Springer, 2007. DOI: 10.1007/11955757_28
[18]. AnimB. Available at: https://wiki.event-b.org/index.php/AnimB
[19]. Code generation activity. Available at: https://wiki.event-b.org/index.php/Code_Generation_Activity
[20]. 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.
[21]. 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.
[22]. A Concise Summary of the Event-B mathematical toolkit. Available at: https://wiki.event-b.org/images/EventB-Summary.pdf
[23]. Петренко А.К., Девянин П.Н., Ефремов Д.В., Карнов А.А., Корныхин Е.В., Кулямин В.В., Хорошилов А.В., Динамическая верификация промышленных средств защиты информации на основе формальных моделей управления доступом. Труды ИСП РАН, 2025, том 37, вып. 3, стр. 275–288. DOI: 10.15514/ISPRAS-2025-37(3)-19 / Petrenko A.K., Devyanin P.N., Efremov D.V., Karnov A.A., Kornykhin E.V., Kulyamin V.V., Khoroshilov A.V., Dynamic verification of industrial information security tools based on formal access control models. Proceedings of the ISP RAS, 2025, vol. 37, issue 3, pp. 275–288. DOI: 10.15514/ISPRAS-2025-37(3)-19

Информация об авторах / Information about authors

Алексей Александрович КАРНОВ – научный сотрудник ИСП РАН. Научные интересы: формальные спецификации, верификация и тестирование, статический и динамический анализ.

Aleksei Aleksandrovich KARNOV – researcher at ISP RAS. Research interests: formal specifications, verification and testing, static and dynamic analysis.

Евгений Валерьевич КОРНЫХИН – кандидат физико-математических наук, доцент кафедры системного программирования МГУ, старший научный сотрудник ИСП РАН. Область интересов: формальная дедуктивная верификация моделей, тестирование на основе моделей.

Eugeny Valerievich KORNYKHIN – Cand. Sci. (Phys.-Math.), Associate Professor of system programming departments at Moscow State University and Senior Researcher at ISP RAS. Fields of Interest: formal deductive verification, model-based testing.