DOI: 10.15514/ISPRAS-2026-38(3)-45

Обзор языковых виртуальных машин и подходов к их тестированию

А.С. Проценко, ORCID: 0009-0001-4240-2986 <protsenko@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Языковые виртуальные машины (ВМ) обычно используются в инфраструктуре систем программирования для объектно-ориентированных языков высокого уровня. Такие языки пользуются популярностью среди разработчиков и исследователей благодаря следующим особенностям: кроссплатформенности, автоматическому управлению памяти (сборке мусора) и изолированной средой исполнения программы, которая, совместно с верификатором кода загружаемых классов, гарантирует определенный уровень безопасности исполняемой программы. В настоящее время существует большое количество как архитектур системы команд ВМ, так и их реализаций. В данном обзоре приводятся список архитектур ВМ и список популярных реализаций ВМ. В статье иллюстрируется схема работы типовой ВМ. Разработка ВМ – сложный процесс, в ходе которого могут совершаться ошибки. Для обеспечения качества реализации ВМ процесс разработки обязательно включает в себя этап тестирования. В обзоре рассматриваются подходы, нацеленные на тестирование ВМ, и производится их сравнение. Рассматривается возможность использования методов тестирования, приведенных в обзоре, для функционального тестирования реализаций существующих и разрабатываемых архитектур ВМ.

Ключевые слова: языковые виртуальные машины; процессные виртуальные машины; виртуальные машины; архитектура системы команд; ISA; виртуальная архитектура системы команд; тестирование; функциональное тестирование.

Для цитирования: Проценко А.С. Обзор языковых виртуальных машин и подходов к их тестированию. Труды ИСП РАН, том 38, вып. 3, часть 4, 2026 г., стр. 37–58. DOI: 10.15514/ISPRAS–2026–38(3)–45.

A Survey of Language Virtual Machines and Approaches to Their Testing

A.S. Protsenko, ORCID: 0009-0001-4240-2986 <protsenko@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Language Virtual Machines (VMs) are commonly used within the infrastructure of programming systems for high-level object-oriented languages. These languages are popular among developers and researchers due to their key features: cross-platform portability, automatic memory management (garbage collection), and an isolated program execution environment. This environment, combined with a bytecode verifier for loaded classes, guarantees a certain level of security for the executed program. Currently, there is a wide variety of both VM instruction set architectures and their implementations. This survey provides a list of such architectures and their most popular implementations. The paper illustrates the operational scheme of a typical VM. However, VM development is a complex process prone to errors. To ensure the quality of a VM implementation, the development process must necessarily include a testing phase. This review examines approaches aimed at testing VMs and provides a comparison of them. The possibility of applying the testing methods discussed in the survey for the functional testing of implementations of existing and emerging VM architectures is considered.

Keywords: language virtual machines; process virtual machines; virtual machines; instruction set architecture; ISA; virtual-ISA; testing; functional testing.

For citation: Protsenko A.S. A Survey of Language Virtual Machines and Approaches to Their Testing. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 3, part 4, 2026, pp. 37-58 (in Russian). DOI: 10.15514/ISPRAS-2026-38(3)-45.

1. Введение

В основе виртуальных машин (ВМ) лежит виртуализация. В соответствии с ГОСТ [1] виртуализация – это группа технологий, основанных на преобразовании формата или параметров программных или сетевых запросов к компьютерным ресурсам с целью обеспечения независимости процессов обработки информации от программной или аппаратной платформы информационной системы. Первые исследования ВМ [2-5] были начаты в 60-х годах прошлого столетия. В это же время были сформулированы основные концепции системы команд (Instruction Set Architecture, ISA), виртуализации и ВМ.

В современных работах [6, 7] ВМ делят на две большие группы. Это системные ВМ (system virtual machine) и процессные ВМ (process virtual machine). Процессные ВМ предназначены для запуска отдельного приложения. Архитектура процессной ВМ представлена на рис. 1. Хост-платформой в процессных ВМ является аппаратное обеспечение совместно с ОС. Программное обеспечение для виртуализации в процессных ВМ часто называют средой исполнения. Гостевая платформа создается для отдельного процесса приложения. Запускаемое приложение представляет собой исполняемый файл в формате понятном для среды исполнения.

Наиболее популярными и известными среди процессных ВМ являются языковые ВМ (language virtual machine). В языковых ВМ используются специальные системы команд, обычно не имеющие аппаратной реализации. Такие системы команд еще называют виртуальными системами команд (Virtual ISA, V-ISA) или байт-кодом. Языковые ВМ обычно используются в инфраструктуре систем программирования для объектно-ориентированных языков высокого уровня. В методологии объектно-ориентированного программирования (ООП) предметная область описывается с помощью иерархии классов. Класс в такой методологии представляет собой шаблон, по которому можно создать объект определенного типа, с заданной структурой и поведением (алгоритмами работы). Информация о классах в

ВМ представляется с помощью метаданных. Языковые ВМ, поддерживающие ООП методологию, обладают следующими свойствами: наличием загрузчика классов и верификатора, наличием интерпретатора байт-кода, работой с библиотеками классов и метаданными, обработкой исключений и использованием сборщика мусора. Далее в работе под ВМ мы будем понимать именно языковые ВМ и их систему команд, если не указано иное.

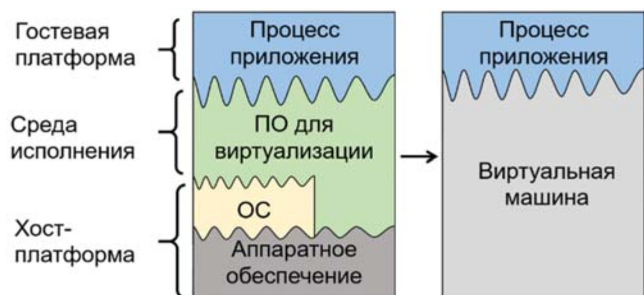

Рис. 1. Архитектура процессной ВМ.
Fig. 1. Process virtual machine architecture.

Во втором разделе статьи приведена общая схема работы ВМ. Рассматриваются особенности ВМ. Приводится список существующих систем команд ВМ и приводятся примеры существующих реализаций. В третьем разделе проводится обзор существующих подходов к тестированию ВМ, проводится сравнения применимости подходов для функционального тестирования как существующих ВМ, так и разрабатываемых. Заканчивается статья заключением.

2. Языковые виртуальные машины

В разделе описываются особенности рассматриваемых в работе ВМ, приводится общая схема работы ВМ, рассматривается стековые и регистровые ВМ. В разделе описываются существующие системы команд ВМ и приводятся примеры их реализаций.

2.1 Особенности языковых виртуальных машин

В классических реализациях ВМ для интерпретации байт-кода используется *интерпретатор*. В этом заключалась причина основного недостатка языковых ВМ, а именно в более медленном исполнении программ, по сравнению с непосредственным исполнением аналогичных программ на аппаратных средствах. Для ускорения исполнения программ на ВМ байт-код транслируют в нативный код целевой архитектуры и исполняют на аппаратной части. Для этого используют следующие подходы:

- JIT-компиляцию (Just-in-Time), которая позволяет применить динамическую компиляцию во время выполнения программы, и
- AOT-компиляцию (Ahead-of-Time), которая применяется до начала выполнения программы.

Программы для языковых ВМ являются *кроссплатформенными*. Они не зависят от операционных систем и микропроцессорных архитектур. Они зависят только от ВМ и если для операционной системы и микропроцессорной архитектуры существует работающая на ней ВМ, то и программу для этой ВМ можно исполнить на этих платформах. Достаточно знаменит слоган “Write once, run anywhere” [8] придуманный в компании Sun Microsystems для демонстрации преимуществ кроссплатформенности языка Java [9], использующего в своей инфраструктуре ВМ.

Непосредственный доступ к памяти и ручная работа с ней часто влечет за собой ошибки. В ВМ для работы с памятью реализован менеджер управления памятью, который снимает с разработчика эту часть задач. Важным механизмом является *сборщик мусора*, который периодически (или по необходимости) запускается и освобождает память от артефактов, которые больше не используются. Эта технология позволяет упростить разработку ПО.

Перед началом работы программы, ее исполняемые файлы необходимо загрузить в ВМ. За стратегию загрузки отвечает *загрузчик классов*, который производит поиск, загрузку и связывание классов. Во время работы загрузчика происходит проверка загружаемых классов на соответствие требованиям с помощью *верификатора*.

Для исполнения программы ВМ создает для нее изолированную среду, что предотвращает доступ к системным ресурсам, которые программе не требуются. При работе программы все взаимодействие с внешней средой осуществляется через ВМ. Такой подход позволяет обеспечивать определенный уровень *безопасности* при исполнении программ с помощью ВМ.

Благодаря своим особенностям ВМ получили широкое распространение, и в настоящее время встречаются повсюду: в сотовых телефонах, ноутбуках, серверах и других электронных вычислительных устройствах.

2.1.1 Общая схема работы виртуальных машин

Общая схема применения ВМ для исполнения программы на языке высокого уровня (ЯВУ) приведена на рис. 2.

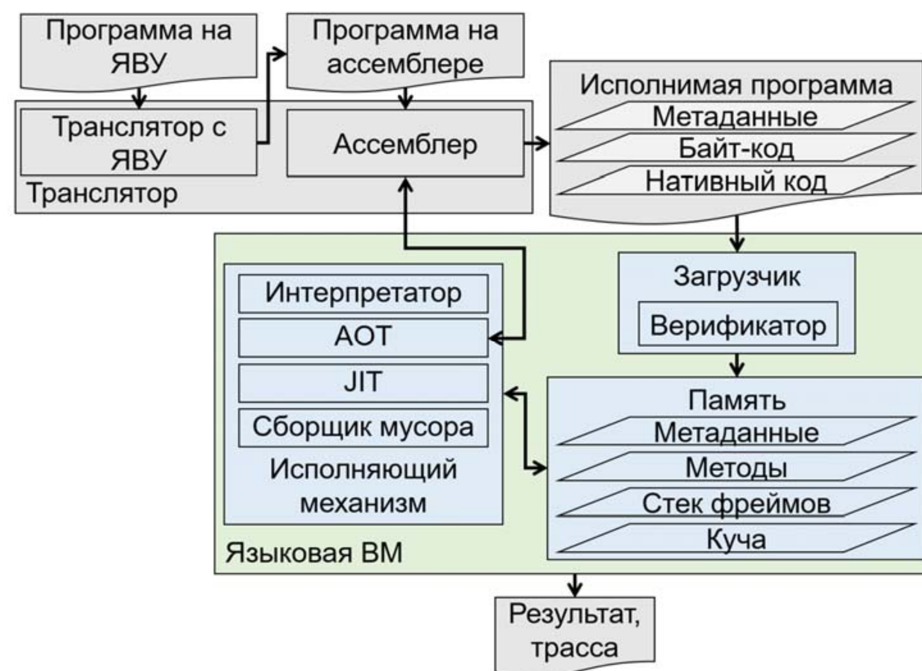

Рис. 2. Общая схема применения ВМ для исполнения программы.
Fig. 2. Program execution scheme using a virtual machine.

Схема состоит из следующих этапов. Сначала программа разрабатывается на ЯВУ, после чего программа на ЯВУ с помощью транслятора с ЯВУ транслируется в программу в текстовом формате на ассемблере. Полученная текстовая программа ассемблируется в исполняемую программу, сохраняемую в исполняемый файл. Исполняемая программа содержит:

- метаданные, с помощью которых описаны классы, поля классов, методы классов, обработчики исключений;
- байт-код, который описывает реализуемое в методах поведение;
- нативный код, в который транслируется байт-код при использовании AOT-компиляции.

Далее исполняемый файл подается ВМ, где верификатор ВМ проверяет его корректность и соответствие требованиям к байт-коду. После этого загрузчик ВМ загружает необходимые метаданные и байт-код в память ВМ и исполняющий механизм начинает исполнять программу.

Основным исполняющим механизмом ВМ является интерпретатор, и именно он обязан быть в состоянии исполнить любой корректный байт-код программы, переданной на исполнение. Во время такого исполнения может создаваться трасса исполнения программы, которая обычно содержит информацию об исполненных инструкциях и изменениях состояния ВМ. Нативный код, полученный при помощи JIT- и AOT-компиляции, исполняется непосредственно на аппаратном обеспечении и, в отличии от режима интерпретации байт-кода, в трассу исполнения программы никакой информации не передает.

Объекты, создаваемые во время исполнения программы, хранятся в области памяти называемой *кучей*. Сборщик мусора позволяет автоматически уничтожать объекты из кучи, которые в программе более не могут быть использованы. При каждом вызове метода создается фрейм, содержащий данные (такие как стек операндов или регистры), необходимые для выполнения байт-кода метода. Создаваемые фреймы сохраняются в стек фреймов. После завершения метода, соответствующий фрейм удаляется из стека фреймов и уничтожается.

После завершения исполнения программы, ВМ возвращает результат и трассу исполнения. В данной статье в качестве результата исполнения программы рассматривается возвращаемое значение. Стоит отметить, что для получения трассы исполнения от ВМ при запуске исполнения программы необходимо передавать в ВМ соответствующие флаги.

Помимо языковых ВМ в некоторой литературе иногда упоминаются названия “ВМ для языков высокого уровня” (High Level Language Virtual Machine, HLL VM) [10] и “ВМ для виртуальной системы команд” (Virtual ISA virtual machine) [7], которые в данной работе рассматриваются как синонимы для языковых ВМ. Такое разнообразие названий для языковых ВМ можно объяснить тем, что в настоящий момент нет устоявшейся терминологии в этой области и наличием исследований по этой тематике от различных научных групп и команд.

2.1.2 Регистровые и стековые виртуальные машины

ВМ можно разделить на две группы по типу используемых операндов в системе команд: регистровые и стековые. *Регистровые* ВМ для передачи операндов и хранении промежуточных значений используют регистры, в то время как *стековые* ВМ для этого используют стек операндов.

При использовании регистров для передачи операндов возрастает размер инструкции, так как в кодировку инструкции, помимо кода инструкции, добавляются индексы используемых регистров. Но в регистровых ВМ нет необходимости каждый раз перезагружать данные в регистре, если до этого он уже был инициализирован.

При использовании стека операндов размеры инструкций не содержат дополнительной информации помимо кода самой инструкции. Но перед использованием целевой операции в стековой ВМ приходится использовать несколько дополнительных операций, которые должны загрузить операнды в стек, преобразование над которыми будет совершать целевая операция, после работы которой использованные данные из стека операндов будут удалены.

В настоящее время ни один из подходов не считается более эффективным в целом и при создании новых ВМ выбор организации работы с операндами связан с особенностями планируемой реализации ВМ. Хотя существуют исследования [11, 12] в которых было показано, что регистровая ВМ при исполнении программ стандартного тестового набора только с помощью интерпретатора затрачивает до 20% времени меньше, чем аналогичная стековая ВМ.

2.2 Системы команд языковых виртуальных машин и их реализации

В настоящее время существует большое количество систем команд ВМ и еще большее количество ВМ их реализующих. Пример популярных и упоминаемых в литературе систем команд ВМ представлены в табл. 1.

Табл. 1. Системы команд и реализации ВМ.
Table 1. Instruction set architectures and VM implementations.

№	Система команд	Тип операндов ВМ	Реализация ВМ	Год создания
1	P-code [13]	Стековая	The P-code Machine [13]	1972
2	PL/EXUS [14]	Стековая	PLEXUS	1973
3	Smalltalk bytecode [15]	Стековая	Smalltalk-80 System [16, 17]	1983
			Squeak [18]	1996
			Pharo [19]	2008
4	Self bytecode [20]	Стековая	Self VM [21, 22]	1987
5	Lua bytecode [23].	Регистровая	Lua [24]	1993
6	K-code [25]	Регистровая	K-machine (Kaleidoscope’93) [25]	1993
7	Python bytecode [26]	Стековая	CPython [27]	1994
8	Java bytecode [28]	Стековая	Kaffe [29]	1996
			Cacao [30]	1997
			Jalapeno VM [31, 32]	1998
			HotSpot [33]	1999
			Jikes RVM (Research Virtual Machine) [34]	1999
			SableVM [35]	2000
			JRockit [36]	2002
			OpenJ9 [37]	2017
			GraalVM [38, 39]	2019
9	SpiderMonkey bytecode [40]	Стековая	SpiderMonkey [41]	1996
10	CIL Instruction Set [42]	Стековая	.NET Framework (Common Language Runtime) [43]	2002
			The Mono Runtime [44]	2004
11	V8 bytecode [45]	Регистровая	V8 engine [46]	2008
12	Dalvik bytecode [47]	Регистровая	Dalvik [48]	2008
			Android Runtime (ART) [49, 50]	2013
13	Parrot bytecode [51]	Регистровая	Parrot [52]	2009
14	Mu Instruction Set [53]	Регистровая	Mu [54]	2015
15	WebAssembly [55]	Стековая	WARDuino [56, 57]	2019
16	Panda bytecode [58]	Регистровая	ARK (static core) [59]	2022

Как видно из табл. 1, для некоторых систем команд ВМ может быть создано несколько реализаций ВМ, чаще всего от различных компаний. При этом для систем команд ВМ могут быть созданы трансляторы с различных языков программирования. Программы на языке JavaScript [60] можно исполнить на таких ВМ как WARDuino, SpiderMonkey и V8. Программы на языке Java можно транслировать в Java байт-код и Dalvik байт-код. При этом для получения Java байт-кода можно использовать программы на таких ЯВУ как Java, Kotlin, Scala, Groovy и др., после чего его можно будет исполнить на одной из реализаций Java Virtual Machine (JVM).

Для языка Python, помимо варианта трансляции в Python байт-код и исполнении на CPython, существуют проекты: Jython [61], позволяющий получать Java байт-код и использовать JVM, и IronPython [62], использующий в своей инфраструктуре .NET и Mono ВМ для исполнения CIL байт-кода.

Такие ВМ как Self, Jalapeno, Cacao и ART не имеют своего интерпретатора и всегда транслируют байт-код в машинный код хост-системы. Реализации Self, Jalapeno и Cacao используют JIT-компиляцию для ускорения выполнения программ. ART ВМ использует AOT-компиляцию, что позволяет добиться ускорения исполнения программ за счет увеличения времени установки приложения в ОС Android [63], во время которой и выполняется трансляция байт-кода в машинный код.

Среди приведенных ВМ существуют реализации, написанные на языке, в инфраструктуре которого они могут быть использованы. К таким ВМ относится GraalVM, написанная на языке Java.

Из современных разработок можно рассмотреть подробнее ВМ ARK (static core), реализующую Panda байт-код. Данная ВМ принимает на вход программы на Panda ассемблере, в которых используется синтаксис байт-кода для описания тел методов класса и специальные ключевые слова для описания классов. После того, как программа на ассемблере транслируется в исполняемый файл, он проверяется с помощью верификатора. Для исполнения программы используется интерпретатор. В ВМ ARK на февраль 2024 года было реализовано 2 интерпретатора: интерпретатор cpp, написанный на языке высокого уровня cpp и интерпретатор irtoc [64], получивший свое название от словосочетания “Ir-To-Code”, который автоматически генерируется из рукописного внутреннего представления. В ВМ ARK реализованы JIT- и AOT-компиляция.

Как видно из табл. 1, существует большое количество систем команд ВМ и их реализаций. При этом сохраняется тенденция к созданию новых систем команд, а реализации ВМ разрабатываются как для новых, так и для существующих систем команд.

3. Тестирование языковых виртуальных машин

В разделе представлена общая схема функционального тестирования и приводятся требования к организации функционального тестирования ВМ. Проводится обзор подходов к тестированию ВМ. Производится оценка соответствия существующих подходов к тестированию ВМ требованиям к функциональному тестированию ВМ.

3.1 Общая схема функционального тестирования

Разработка ВМ – сложный процесс, в ходе которого могут совершаться ошибки. Для обеспечения качества реализации ВМ процесс разработки включает в себя этап функционального тестирования. *Функциональное тестирование* программы – это исследование программы для выявления отличий между ее реально существующим поведением и требуемым поведением в ситуациях из выделенного конечного набора [65, 66]. Общая схема функционального тестирования представлена на рис. 3.

Сформулируем список основных элементов схемы, которые необходимы для организации функционального тестирования ВМ:

- **спецификация**, которая содержит информацию об эталонном поведении системы;
- **тестируемая система** – реализация, которая проверяется на соответствие спецификации;
- **тестовые программы**, с помощью которых осуществляется тестовое воздействие на тестируемую систему;
- **оракул**, который проводит проверку результатов тестового воздействия;
- **критерий тестового покрытия** – метрика, позволяющая оценить качество тестирования.

Элементы, представленные на схеме рис. 3, являются абстрактными понятиями. Так, например, спецификацией может быть, как документация на тестируемую ВМ (в таком случае разработка тестовых программ осуществляется вручную), так и описание на формальном языке, где для разработки тестовых программ используются генераторы на основе формального описания.

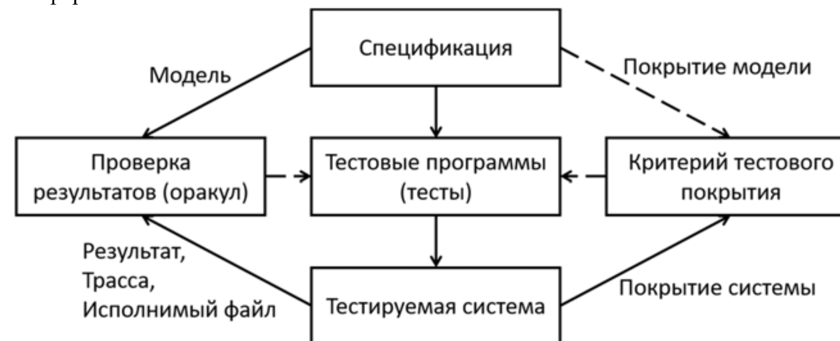

Рис. 3. Общая схема функционального тестирования.
Fig. 3. General scheme of functional testing.

При организации функционального тестирования важно ответить на следующие вопросы:

1) Как создавать тестовые программы?
2) Как проверять результат исполнения тестовой программы?
3) Как оценивать качество тестирования?

3.2 Обзор подходов к тестированию виртуальных машин

В подразделе представлен обзор работ из открытых источников. Рассматривались работы, в которых описаны подходы, применяемые или которые могут быть полностью или частично применены к тестированию ВМ. В рассматриваемых работах основное внимание уделялась следующим вопросам:

1) Как создавались тестовые программы?
   a) Как предоставлялась информация о формате тестовой программы?
   b) Какие техники создания тестовых программ использовались?
   c) Какая информация о тестируемой системе использовалась?
2) Как проверялись результаты исполнения тестовой программы?
3) Как оценивалось качество тестирования?

Рассмотрим работы, посвященные тестированию ВМ.

В статье [67] авторы из США предлагают подход для тестирования JVM, в основе которого лежат порождающие грамматики. В работе описывается предметно-ориентированный язык lava, применяемый для описания порождающих грамматик, с помощью которых генерируются тестовые программы. Для проверки результата исполнения тестовой программы в статье используются два подхода: дифференциальное тестирование [72] и оракул на основе сертификатов. Для дифференциального тестирования в статье используются следующие реализации JVM: Sun JDK 1.0.2 [68] и Microsoft JVM из Internet Explorer 4.0 [69]. Сертификат представляет собой краткое описание ожидаемого поведения системы для заданной тестовой программы, он создается на основе специального расширения производственных грамматик, доступного в lava. Для создания исполняемых файлов из программ на байт-коде в работе используется инструмент Jasmin [70, 71].

При *дифференциальном тестировании* случайно сгенерированные тестовые программы подаются двум и более сопоставимым системам [72]. В более поздних работах по тестированию ВМ авторы под применением дифференциального тестирования чаще всего понимают фаззинг подход для создания тестовых программ и использование нескольких реализаций ВМ для оценки результатов тестовых воздействий. В оригинальной статье автор William M. McKeeman, хоть и предлагал использовать случайную генерацию для создания большого количества тестовых программ, обращал свое внимание на то, что для того, чтобы сделать дифференциальное тестирование эффективным, нужно улучшать качество тестов. В работах некоторых авторов для создания тестовых программ используется не случайная генерация, но для оценки результатов тестовых воздействий используются две и более сопоставимых систем, именно такую оценку они называют дифференциальным тестированием. Чтобы отличить оригинальный подход дифференциального тестирования от части подхода для оценки результатов тестирования в обзоре иногда будет использоваться понятие *дифференциального сравнения*, под которым понимается сравнение результатов исполнения тестовых программ на двух разных реализациях ВМ, версиях ВМ или механизмах исполнения программ. При использовании дифференциального сравнения считается, что в одной из систем содержится ошибка, если результаты исполнения программ на разных реализациях расходятся.

В работе [73] авторы предыдущей статьи [67] продолжают исследования, направленные на тестирование JVM, и предлагают, помимо использования порождающих грамматик, применять фаззинг тестирование. Для создания тестовых программ использовались вручную написанные тестовые базы, к которым применялись однобайтовые случайные мутации кода. Для дифференциального сравнения используются следующие реализации JVM: Sun JDK 1.0.2, Microsoft JVM из Internet Explorer 4.0 и Netscape 4.0 [74].

В работе [75] исследователи из Fujitsu предлагают подход случайной генерации тестовых программ для JIT компилятора JVM. Для создания тестовых программ вначале случайным образом генерируются классы, методы классов и поля классов со случайными именами и модификаторами доступа. Классы соответствуют ациклическому графу иерархии. Методы классов разделяются на разные уровни и при генерации тестовых программ методы классов могут вызывать только методы классов более низкого уровня. Далее строится граф потока управления и ограничений по данным, в соответствии с которым тела методов классов заполняются байт-код инструкциями, среди которых могут использоваться последовательности, которые взаимодействуют с полями классов или вызывают методы классов более низкой иерархии. Во избежание зацикливания, все циклы снабжаются счетчиком и инструкцией выхода из цикла, срабатывающей при достижении счетчиком определенного ранее значения. Для проверки результатов исполнения тестовой программы в них встраиваются операции вывода состояния некоторых переменных и полей классов. Значения этих переменных сравниваются после исполнения тестовых программ с использованием тестируемого JIT компилятора и после исполнения этих же программ в других режимах или на других версия JVM.

В статье [76] авторы представляют метод генерации тестовых программ на основе комбинаторного перебора пар байт-код инструкций JVM. Идея такого перебора основана на том, что операнды инструкций должны иметь определенный тип, и для каждого типа есть специальные инструкции загрузки данных в стек операндов, загрузки данных в стек локальных переменных и так далее. В своей работе авторы разделяют байт-код инструкции по группам, в соответствии с типами операндов. Далее на основе этих групп они создают некорректные комбинации инструкций. Цель создания таких комбинаций – это проверка реализации JVM на типобезопасность. Полученные комбинации инструкций проверяют методом проверки моделей на формальной модели JVM, описанной на NuSMV [77], и получают информацию о корректности таких комбинаций. С помощью BCEL [78] создают программы на байт-коде и затем проверяют их с помощью встроенного верификатора. Результаты, полученные от верификатора, сравнивают с результатами, полученными от модели JVM на NuSMV. Результаты от обоих источников должны совпадать.

В работе [79] авторы представляют инструмент DexFuzz, реализующий метод дифференциального тестирования на основе бинарного фаззинга байт-кода DEX (Dalvik Executable). В качестве исходных программ для мутаций взят тестовый набор для ART ВМ, состоящий из 200 тестов. Оценка результатов исполнения тестов производится с помощью дифференциального тестирования, где тестовые программы исполняются с помощью бэкендов (backends) ART ВМ: интерпретатора, быстрого AOT компилятора и оптимизирующего AOT компилятора.

В статье [80] предложен метод фаззинг тестирования ВМ, реализованный в инструменте ClassFuzz. В работе для создания тестовых программ используется фаззинг, ориентированный на увеличения покрытия кода JVM. Для этого в инструменте было реализовано 129 операций мутации. Для внедрения мутаций в Java байт-код применяются преобразования над Jimple [81]. Jimple – промежуточное представление фреймворка Soot [82], используемое для анализа Java классов и получение их представление в байт-код подобном виде. Для изменения исполнимых файлов JVM *.class на уровне байт-кода создано 6 мутаций, остальные 123 мутации нацелены на изменения на уровне высокоуровневого языка и могут, например, менять атрибуты классов, добавлять в класс интерфейсы или возможные исключения. В качестве зерен для мутаций были взяты 1216 классов из JRE7 библиотек. Результаты исполнения программ оцениваются с помощью дифференциального сравнение, где в качестве ВМ используются следующие реализации: HotSpot для версий Java 7/8/9, J9 для IBM SDK8 [83] и GIJ 5.1.0 [84].

В работе [85] описан подход дифференциального тестирования JVM. Подход реализован в инструменте Classming. Для генерации тестовых программ используется фаззинг. В качестве зерен для мутаций используется тестовый набор DaCapo [86] для языка Java. В подходе используются мутации байт-кода метода, направленные на изменения потока данных и потока управления. Для изменения исполнимых файлов используется фреймворк Soot и его промежуточное представление для байт-кода Jimple. Для внедрения мутаций в метод используются следующие 5 инструкций: `goto`, `return`, `throw`, `lookupswitch`, `tableswitch` доступные в Jimple. Результаты исполнения тестовых программ оцениваются с помощью дифференциального сравнения, где в качестве ВМ используются следующие реализации: HotSpot из инструментария OpenJDK [87] и J9 от IBM [88], в настоящее время известная как OpenJ9.

В статье [89] описан подход для тестирования ВМ с интерпретатором байт-кода, основанный на конколическом тестировании (concolic testing). Конколическое тестирование – это гибридная техника тестирования, объединяющая в себе конкретное (concrete) и символическое (symbolic) выполнение для автоматической генерации тестовых случаев [90]. Авторы применяют конколическое тестирование к интерпретатору ВМ для получения списка значений для покрытия всех возможных путей исполнения интерпретатора. На основе полученных данных формируется тестовый набор, который подается для исполнения на ВМ

в двух режимах: интерпретатора и JIT-компилятора. Для оценки результатов используется дифференциальное сравнение. Схема тестовой системы, применяемой в работе, описана в работе [91] и базируется на использовании модульного тестирования (unit testing). Подход был применен к 4 различным версиям JIT-компиляторов Pharo ВМ, в работе которых были найдены расхождения.

В диссертации [92] описан подход для проверки семантической корректности сгенерированных ВМ с помощью среды генерации ВМ Slang на основе моделирования (simulation-based VM generator framework). Генератор ВМ Slang принимает в качестве входных данных описание ВМ на языке Pharo, а в качестве выходных данных возвращает реализацию ВМ на языке C. Для проверки семантической корректности используется дифференциальное сравнение результатов исполнения тестовых программ на сгенерированной ВМ и модели из среды генерации ВМ Slang. Сравниваемыми результатами работы программ являются корректное завершение программы или ошибка. Ошибки, возникающие в тестовых программах, могут быть следующих типов: результат работы программы не совпадает с эталонным, генерация исключения во время выполнения программы и ошибка во время компиляции программы. В работе описано использование двух видов тестовых наборов. Первый тестовый набор является рукописным, который использовался для отладки модели ВМ в среде генерации. Второй набор был получен путем мутаций рукописных тестов. С помощью описанного подхода были найдены семантические расхождения при работе ВМ, что повлекло внесение правок в генератор ВМ Slang.

В работе [93] авторы представляют инструмент JavaTailor, являющийся файзером для JVM. Метод, лежащий в основе инструмента, описывается с помощью трех этапов: извлечение "ингредиентов" из набора тестовых программ, с помощью которых были ранее найдены ошибки; генерация тестовых программ на основе ингредиентов и зерна (seed) в виде класса на языке Java; проверка результата исполнения программы с помощью дифференциального тестирования. Для дифференциального тестирования в работе использовались OpenJ9 с SDK (Software Development Kit) от IBM [94] и HotSpot разных версий. Авторы описывают свой подход как направленный на создание тестовых программ для поиска ошибок, в то время как подходы, реализованные в инструментах ClassFuzz и Classming, по их мнению, нацелены на создание тестовых программ с разнообразным потоком управления и данных путем накопления незначительных мутаций.

В статье [95] описан фреймворк JITfuzz, реализующий метод фаззинг тестирования, ориентированный на покрытие кода JIT компилятора JVM. Авторы подчеркивают, что из-за особенностей работы JIT компилятора, обычными мутациями программ сложно достичь высокого покрытия его кода. В работе предложены 4 мутации, нацеленные на активацию оптимизаций, и 2 мутации, нацеленные на обогащение графа потока управления. Для внедрения мутаций в Java байт-код применяются преобразования над Jimple. Для гарантированного включения JIT оптимизации Java класса используется специальная опция JVM. В качестве начальных зерен для мутаций используются программы на Java из открытых проектов и специальных тестовых наборов для тестирования JVM. Для проверки результатов исполнения программ в работе применяется дифференциальное тестирование. В качестве ВМ используются HotSpot и OpenJ9.

В работе [96] представлен инструмент SJFuzz, являющийся фаззером для JVM. Авторы характеризуют свой инструмент как более управляемый, чем инструменты ClassFuzz и Classming, и, как следствие, дающий лучшие результаты, в том числе и по сравнению с JavaTailor. Для внедрения мутаций в Java байт-код применяются преобразования над Jimple. Для управление графом потока управления используются инструкции: `goto`, `lookupswitch` и `return`. Авторы предлагают несколько алгоритмов, позволяющих разнообразить мутации и уменьшить количество мутаций, приводящих к одинаковым ошибкам. Например, для сравнения тестовых программ между собой используется метрика, основанная на сравнении инструкций метода, которые были исполнены. Для оценки

результатов используется дифференциальное тестирование, а в качестве ВМ используются HotSpot, DragonWell от Alibaba [97], OpenJ9, Zulu от Azul [98] и интерпретатор GIJ GNU.

Тестирование компиляторов для языков высокого уровня. Помимо работ, посвященных непосредственно тестированию ВМ, существует ряд работ, в которых описаны подходы, которые полностью или частично можно применить к тестированию ВМ. Рассмотрим такие работы.

В статье [99] описывается инструмент Csmith, предназначенный для тестирования компиляторов для языка программирования C [100]. Генератор Csmith случайным образом генерирует программы на языке C на основе требований грамматики и ограничений стандарта. Полученные программы компилируются с помощью разных компиляторов и для разных микропроцессорных архитектур. Далее скомпилированные исполняемые файлы запускаются. Для оценки результата исполнения полученных программ используется дифференциальное сравнение.

В работе [101] представлен подход SPE (Skeletal Program Enumeration) для тестирования компиляторов. Идея подхода заключается в следующем. Из программы можно получить синтаксический каркас, где вместо имен используемых переменных образуются пропуски, которые нужно заполнить, и список переменных, которые можно использовать для заполнения каркаса. Заполняются пропуски именами переменных таким образом, чтобы перебрать все возможные варианты использования переменных в синтаксисе каркаса. Авторы пишут, что на основе анализа сообщений об ошибках в репозиториях компиляторов GCC [102] и Clang [103] было подсчитано, что минимальные тестовые программы, выявляющие ошибки, состоят из порядка 30 строк кода. А после анализа тестового набора c-torture [104] для GCC-4.8.5 исследователи подсчитали, что каждая функция в среднем содержит только 3 переменные с 7 местами, где она используется. То есть перебор всех возможных вариантов использования переменных в создаваемых синтаксических каркасах возможно произвести за относительно небольшое время. При этом из групп семантически эквивалентных программ необходимо оставить только по одному экземпляру. Исследователи использовали тестовый набор для GCC-4.8.5 в качестве основы для получения синтаксических каркасов и тестировали две стабильные версии компиляторов GCC-4.8.5 и Clang-3.6.1. Для анализа результатов исполнения тестовых программ использовалось дифференциальное сравнение. Применение данного подхода позволило обнаружить более 217 ошибок.

В статье [105] представлен инструмент JAttack, разработанный для тестирования компиляторов языка Java. Для создания тестовых программ на языке Java инструмент JAttack использует шаблоны на предметно-ориентированном языке (Domain-Specific Language, DSL) Java, в которых можно обозначить пропуски, которые во время генерации тестов заполняются случайными выражениями и значениями, доступными в пространстве поиска, определяемом пропуском. В статье описано использование шаблонов, разработанных вручную и полученных на основе проектов с открытым исходным кодом. Помимо тестирования компилятора с языка Java, инструмент был применен для тестирования JIT компилятора. Для проверки результатов в работе применяется дифференциальное сравнение и используются следующие комплекты разработчика приложений для языка Java версии 11.0.8: Oracle JDK [106], OpenJDK и OpenJ9. С помощью инструмента было найдено 6 ошибок, подтвержденных разработчиками из компании Oracle.

В диссертации [107] представлен подход для фаззинг тестирования компиляторов для языка Kotlin. Для генерации тестовых программ используется подход типо-ориентированной генерации по шаблону, являющийся развитием идеи из работы [101]. В качестве основы для создания тестовых программ используется набор программ на языке Kotlin. Из набора берется несколько программ: одна программа используется как основа для создания шаблона, вторая используется как источник для внесения мутаций в шаблон. После слияния программ они трансформируются в синтаксический шаблон, содержащий ячейки для заполнения в

местах использования переменных. Для заполнения ячеек используется множество доступных в области видимости переменных и сгенерированные случайные выражения, имеющие совместимый тип. Для оценки результатов тестирования используется дифференциальное сравнении, в котором используются различные реализации, версии и режимы запуска компиляторов.

Рукописные тестовые наборы. Для тестирования и оценки характеристик ВМ применяются рукописные тестовые наборы. Такие тестовые наборы обычно нацелены на разные механизмы и специфицированное поведение проверяемых ВМ, что делает их интересным источником для создания новых тестовых программ. Примеры таких наборов представлены ниже.

Для оценки характеристик производительности ВМ Java разработан тестовый набор DaCapo [86, 108]. Он содержит используемые сообществом разработчиков программы с открытым исходным кодом, содержащими нетривиальные нагрузку на память.

Для обеспечения совместимого поведения между реализациями платформы Java SE (Standard Edition) используется набор тестов Java Compatibility Kit (JCK) [109]. Данный тестовый набор создан на основе требований JSR (Java Specification Request) к платформе Java.

Некоммерческая корпорация SPEC (Standard Performance Evaluation Corporation) предлагает для оценки производительности ВМ Java и используемых аппаратных систем многопоточный тестовый набор SPECjvm2008 [110]. Он содержит несколько реальных приложений и тестовых наборов, нацеленные на основные функциональные возможности Java платформы.

3.3 Анализ подходов к тестированию виртуальных машин

На основе проведенного обзора можно выделить следующие **техники создания тестовых программ**:

- ручная разработка [73, 92, 108, 109, 110];
- автоматическая или автоматизированная генерация тестов:
  - порождающие грамматики [67];
  - комбинаторная генерация [76, 101];
  - случайная генерация [75, 105];
  - конколическая генерация [89];
  - фаззинг на основе мутаций [73, 79, 80, 85, 92, 93, 95, 96, 107].

При создании тестовых программ важно представить их в формате, который пройдет проверки компилятора и верификатора. Для этого нужна дополнительная информация о бинарном или текстовом формате тестовых программ. На основе проведенного обзора литературы составим список источников, используемых для получения информации о **формате тестовой программы**:

- документация на ВМ [108-110];
- шаблоны [75, 76, 105];
- корпуса программ [73, 79, 80, 85, 92, 93, 95, 96, 101, 107].

При создании тестовых программ необходима информация о тестируемой системе, которая, например, может быть использована для формирования итогового вида тестовой программы или использоваться в техниках генерации. На основе проведенного обзора составлен список источников, содержащих **информацию о тестируемой системе**:

- документация на ВМ [73, 92, 108-110];
- правила грамматики [67];

- модели [89];
- описание инструкций [76, 75];
- описание мутаций [79, 80, 85, 93, 95, 96, 107].

В некоторых подходах [89] техники генерации тестовых программ могут использоваться совместно с моделью, позволяющей отслеживать состояние системы, для построения сложных тестовых последовательностей, проверки корректности генерируемых программ или управления генерацией, в зависимости от покрытия.

На основе проведенного обзора был составлен список вариантов представления тестовых программ:

- ассемблерный код, содержащий сгенерированные байт-код и метаданные;
- исполняемый файл, содержащий сгенерированные байт-код и метаданные;
- программа на языке высокого уровня.

Варианты используемых техник создания тестовых программ, вариантов предоставления информации о тестируемой системе и формате тестовых программ довольно разнообразны. Хотя в последние годы фаззинг на основе мутаций стал наиболее популярным подходом, встречающимся в работах посвященным тестированию ВМ.

После создания тестовой программы и ее исполнения не тестируемой системе, необходимо оценить результат исполнения тестовой программы. Для этого создаются специальные артефакты, называемые оракулами. Термин *тестовый оракул* (test oracle) впервые был использован Уильямом Хауденом в 1978 [111]. Тестовый оракул – это механизм, позволяющий оценить результат тестового воздействия. Оракул сравнивает эталонный (корректный) результат исполнения тестовой программы с результатом от реализации, полученного при исполнении тестовой программы на тестируемой системе, что дает нам информацию о наличии расхождений. Если расхождения нет, то это говорит об отсутствии обнаруженных ошибок. Если есть расхождение, то говорится о нахождении ошибки.

На основе ранее проведенного обзора литературы составим список применяемых **подходов для создания тестовых оракулов**:

- ручное описание [108-110] (создание детерминированных оракулов с закодированными эталонными значениями);
- дифференциальное сравнение [67, 72, 73, 75, 79, 80, 85, 89, 92, 93, 95, 96, 105, 101, 107];
- оценка результатов (возвращаемого значения) работы программы с ожидаемым результатом, полученным в ходе создания тестовой программы [67, 76].

Из обзора видно, что наиболее популярным является дифференциальное сравнение. Это объясняется низкими затратами на создание оракула таким способом.

В рассмотренных работах для **оценки тестового покрытия** использовалась оценка покрытия тестируемой системы. Оценка тестового покрытия спецификаций, в рассмотренных работах, не предлагалась. Хотя в работе [89] авторы называют реализацию интерпретатора – исполнимой спецификацией, используют ее для генерации тестовых программ для JIT компилятора и покрытие кода интерпретатора достигается благодаря применяемому колколическому подходу.

3.4 Требования к методу функционального тестирования виртуальных машин

На основе проведенного анализа предметной области, общей схемы функционального тестирования, рассмотренных типов существующих ВМ можно сформулировать ряд

требований к методу функционального тестирования ВМ для объектно-ориентированных языков программирования.

Общие требования. Метод должен:

1) Осуществлять тестирование системы как “черного ящика”.
2) Использовать **спецификацию ВМ**, для получения информации о тестируемой системе.
3) Быть применимым к новым системам команд ВМ.
4) Использовать **тестовые программ** для воздействия на целевую систему.
5) Иметь возможность применения на ранних этапах разработки ВМ.
6) Иметь возможность **проверять результаты тестирования**.
7) Позволять оценивать **тестовое покрытие**.

Для проверки функциональных требований с помощью функционального тестирования необходимы знания о спецификации ВМ. Для автоматизации использования документации на систему команд ВМ создают формальные спецификации ВМ. Составим список требований к таким спецификациям. Спецификации должны:

1) Описывать систему команд ВМ.
2) Позволять описывать стековые и регистровые системы команд ВМ.
3) Предоставлять информацию о функциональных свойствах ВМ.
4) Настраивать метод на работу с целевой системой команд ВМ.

При функциональном тестировании ВМ в режиме “черного ящика” воздействие на систему необходимо осуществлять с помощью тестовых программ. Тестовые программ должны:

1) Не зависеть от существующих корпусов программ для целевой ВМ.
2) Создаваться на байт-коде (ассемблере) целевой ВМ. Именно программы на байт-коде позволяют покрыть все функциональные возможности ВМ.
3) Иметь возможность описывать метаданные класса.
4) Использовать тестовые шаблоны для описания формата тестовой программы и повторяющихся общих частей.
5) Иметь возможность нацеливаться на указанные группы инструкций или механизмы.

Оракул, осуществляющий проверку результатов тестирования, должен:

1) Осуществлять проверку результатов тестирования при отсутствии эталонной реализации ВМ.
2) Осуществлять проверку трассы исполнения тестовой программы.
3) Осуществлять проверку трассы исполнения тестовой программы, полученной с применением JIT- и AOT-компиляции.

3.5 Анализ соответствия требованиям существующих подходов

В табл. 2 представлено соответствие требованиям из подраздела 3.4 существующих подходов тестирования ВМ из подраздела 3.2.

Рассмотрим требование использования спецификаций. Этому требованию удовлетворяют два из рассматриваемых подходов: конколическое тестирование, предложенное Guillermo Polito, и ручная разработка. При этом для конколического тестирования в качестве спецификаций используется реализация интерпретатора, которую автор подхода называет “исполнимой спецификацией”, что нарушает требование к тестированию системы как черного ящика. При ручной разработке спецификации тестируемой системы изучаются инженерами-тестировщиками, что позволяет им использовать эти знания при написании

тестовых программ, но такой подход невозможно автоматизировать. Как видно из табл. 2, ни один из приведенных подходов не позволяет специфицировать систему команд ВМ.

Табл. 2. Соответствие требованиям существующих подходов.
Table 2. Compliance of existing approaches with requirements.

Группа	Общее					Специфи-кация		Тестовые программы			Оракул		Покры-тие
№	1	2	3	4	5	6	7	8	9	10	11	12	13
Ограничения и требования \ Инструменты и технологии	Система команд (ISA)	Трудоемкость	Уровень автоматизации	Применимость к новым ISA ВМ	Применимость на ранних этапах разработки ВМ	Использование спецификации	Возможность специфицировать ISA ВМ	Программа на байт-коде	Не зависит от корпусов программ	Работа с метаданными	Проверка возвращаемого значения	Проверка трассы	Оценка критерия тестового покрытия
Порождающие грамматики (Sirer)	Java	Ср.	Ср.	Нет	Нет	Нет	Нет	Да	Да	Нет	Да	Нет	Нет
Случайная генерация (Yoshikawa)	Java	Низ.	Выс.	Нет	Нет	Нет	Нет	Да	Да	Да	Да	Нет	Нет
Комбинаторный перебор (Calvagna)	Java	Ср.	Ср.	Нет	Да	Нет	Нет	Да	Да	Нет	Да	Нет	Нет
Бинарный фаззинг: DexFuzz	Dalvik	Низ.	Выс.	Да	Да	Нет	Нет	Да	Нет	Да	Да	Нет	Нет
Фаззинг: ClassFuzz	Java	Низ.	Выс.	Нет	Нет	Нет	Нет	Да	Нет	Да	Да	Нет	Нет
Фаззинг: Classming, JavaTailor, SJFuzz	Java	Низ.	Выс.	Нет	Нет	Нет	Нет	Да	Нет	Нет	Да	Нет	Нет
Конколическое тестирование (Polito)	Smalltalk	Низ.	Выс.	Нет	Нет	Част.	Нет	Да	Да	Да	Да	Нет	Част.
Рукописные тестовые наборы	Smalltalk, Java	Выс.	Низ.	Да	Да	Да	Нет	Нет	Да	Да	Да	Нет	Нет
Фаззинг: JITfuzz	Java	Низ.	Выс.	Нет	Нет	Нет	Нет	Да	Нет	Нет	Да	Нет	Нет

Следующая группа требований связана с использованием тестовых программ для воздействия на тестируемую систему. За исключением рукописных тестовых наборов, все остальные приведенные в табл. 2 подходы используют тестовые программы на байт-коде. При этом для возможности применения подхода к новым системам команд ВМ необходимо отсутствие зависимости от корпусов программ, которые в некоторых подходах используются как основы для создания тестовых программ. Для создания разнообразных тестовых программ, охватывающих многие особенности ВМ, необходима возможность манипуляций с метаданными классов в тестовых программах, которая присутствует не во всех рассматриваемых подходах.

Все подходы, приведенные в табл. 2, позволяют оценивать результат тестового воздействия. В большинстве подходов для этого используется дифференциальное сравнение, где в качестве эталонной ВМ используется реализация от сторонних разработчиков. Что делает такие подходы не применимыми для проверок новых ISA ВМ, для которых не существует

реализаций от сторонних разработчиков. Подходы, использующие для оценки результатов тестирования отличные от дифференциального сравнения механизмы, могут быть применены к новым системам команд ВМ. К таким подходам относится конколическое тестирование, ручная разработка и подход на основе комбинаторного перебора от Andrea Calvagna и Emiliano Tramontana. При этом из всех рассмотренных нигде не используется сравнение трасс. Именно сравнение трасс позволяет оценить корректность исполнения каждой инструкции тестовой программы.

Часть рассмотренных подходов использует критерий тестового покрытия в фаззинг тестировании для оценки полученных мутантов и управления мутациями. В подходе на основе конколичекого анализа генерируемые тестовые программы должны покрывать все пути исполнения инструкций по умолчанию. Но у большинства из рассмотренных подходов отсутствуют какие-либо встроенные механизмы для оценки критерия тестового покрытия спецификации. При этом для оценки критерия тестового покрытия достаточно часто используют покрытие кода тестируемой системы с помощью сторонних специальных инструментов сбора покрытия.

На основе проведенного анализа соответствия требованиям существующих подходов можно сделать вывод о том, что существующие подходы лишь частично соответствуют предъявляемым требованиям.

Заключение

В статье была рассмотрена общая схема работы языковых ВМ, описаны различия между стековой и регистровой ВМ. Был проведен обзор существующих систем команд ВМ и приведены примеры наиболее популярных ВМ их реализующих. Из обзора видно, что тенденция к разработке как новых систем команд ВМ, так и их реализаций, по-прежнему сохраняется.

В работе представлен обзор существующих подходов тестирования ВМ и сформулированы требования к функциональному тестированию ВМ. Показано, что существующие подходы к тестированию ВМ лишь частично соответствуют предложенным требованиям к функциональному тестированию ВМ. А именно: отсутствует возможность использовать спецификации ВМ и возможность специфицировать новые системы команд ВМ; большинство предложенных подходов к созданию тестовых программ используют существующие корпусы программ, а для оценки результата тестового воздействия используют дифференциальное сравнение, что делает невозможным применение таких подходов к ВМ реализующие новые системы команд; отсутствует оценка тестового покрытия спецификаций ВМ. Все это делает разработку подхода к функциональному тестированию ВМ, соответствующего предложенным в работе требованиям, актуальной задачей.

Список литературы / References

[1]. ГОСТ Р. 56938-2016. Защита информации. Защита информации при использовании технологий виртуализации. Общие положения.-М.: Стандартинформ. 2016.
[2]. Amdahl G.M., Blaauw G.A., Brooks F.P. Architecture of the IBM System/360. IBM Journal of Research and Development, 1964, vol. 8, no. 2, pp. 87-101.
[3]. Anderson D.W., Sparacio F.J., Tomasulo R.M. The IBM System/360 model 91: Machine philosophy and instruction-handling. IBM Journal of Research and Development, 1967, vol. 11, no. 1, pp. 8-24.
[4]. Goldberg R.P. Architecture of virtual machines. Proceedings of the workshop on virtual computer systems, 1973, pp. 74-112.
[5]. Goldberg R.P. Survey of virtual machine research. Computer, 1974, vol. 7, no. 6, pp. 34-45.
[6]. Smith J., Nair R. Virtual machines: versatile platforms for systems and processes. Elsevier, 2005.
[7]. Li X.F. Advanced design and implementation of virtual machines. CRC Press, 2016.
[8]. Описание слогана “Write once, run anywhere”. Доступно по ссылке: https://en.wikipedia.org/wiki/Write_once,_run_anywhere. Дата обращения 10.02.2025.

[9]. The Java Language Specification. Available at: https://docs.oracle.com/javase/specs/jls/se11/html/index.html, accessed 14.02.2025.
[10]. Durelli V.H.S., Felizardo K.R., Delamaro M.E. Systematic mapping study on high-level language virtual machines. Virtual Machines and Intermediate Languages, 2010, pp. 1-6.
[11]. Shi Y. et al. Virtual machine showdown: Stack versus registers. ACM Transactions on Architecture and Code Optimization (TACO), 2008, vol. 4, no. 4, pp. 1-36.
[12]. Fang R., Liu S. A performance survey on stack-based and register-based virtual machines. arXiv preprint arXiv:1611.00467. 2016.
[13]. Pemberton S., Daniels M. Pascal implementation. Ellis Horwood, 1982.
[14]. Sitton G.A., Kendrick T.A., Carrick A.G. The PL/EXUS language and virtual machine. Proceedings of a symposium on High-level-language computer architecture, 1973, pp. 124-130.
[15]. Goldberg A., Robson D. Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc., 1983.
[16]. Deutsch L.P., Schiffman A.M. Efficient implementation of the Smalltalk-80 system. Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1984, pp. 297-302.
[17]. Xerox 1100 Scientific Information System. Available at: https://archive.computerhistory.org/resources/access/text/2010/06/102660634-05-03-acc.pdf, accessed 19.02.2025.
[18]. Squeak Virtual Machine. Available at: https://squeak.org/, accessed 14.02.2025.
[19]. Pharo Virtual Machine. Available at: https://github.com/pharo-project/pharo-vm, accessed 14.02.2025.
[20]. CHAMBERS C., UNGAR D. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. ACM SIGPLAN notices, 2004, vol. 39, no. 4, pp. 298-312.
[21]. The Self Handbook. Available at: https://handbook.selflanguage.org/2024.1/, accessed 18.02.2025.
[22]. Язык программирования Self. Available at: https://selflanguage.org/, accessed 18.02.2025.
[23]. Lua 5.3 Bytecode Reference. Available at: https://the-ravi-programming-language.readthedocs.io/en/latest/lua_bytecode_reference.html, accessed 14.02.2025.
[24]. Ierusalimschy R., De Figueiredo L.H., Celes Filho W. The Implementation of Lua 5.0. J. Univers. Comput. Sci., 2005, vol. 11, no. 7, pp. 1159-1176.
[25]. Lopez G., Freeman-Benson B., Borning A. Implementing constraint imperative programming languages: the Kaleidoscope'93 virtual machine. Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications, 1994, pp. 259-271.
[26]. Disassembler for Python bytecode. Available at: https://docs.python.org/3/library/dis.html#python-bytecode-instructions, accessed 18.02.2025.
[27]. Ike-Nwosu O. Inside the Python Virtual Machine, 2015.
[28]. Java Language and Virtual Machine Specifications. Available at: https://docs.oracle.com/javase/specs/index.html, accessed 14.02.2025.
[29]. The Kaffe Virtual Machine. Available at: https://github.com/kaffe/kaffe, accessed 14.02.2025.
[30]. Krall A. Efficient JavaVM just-in-time compilation. Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192, IEEE), 1998, pp. 205-212.
[31]. Alpern B. et al. Implementing Jalapeno in Java. ACM SIGPLAN Notices, 1999, vol. 34, no. 10, pp.314-324.
[32]. Alpern B. et al. The Jalapeno virtual machine. IBM Systems Journal, 2000, vol. 39, no. 1, pp. 211-238.
[33]. The Java HotSpot Performance Engine Architecture. Available at: https://www.oracle.com/java/technologies/whitepaper.html, accessed 14.02.2025.
[34]. Jikes RVM. Available at: https://www.jikesrvm.org/, accessed 17.02.2025.
[35]. Gagnon E.M., Hendren L.J. {SableVM}: A Research Framework for the Efficient Execution of Java Bytecode. Java (TM) Virtual Machine Research and Technology Symposium (JVM 01), 2001.
[36]. About the Oracle JRockit JDK. Available at: https://docs.oracle.com/cd/E15289_01/JRSDK/aboutjrockit.htm, accessed 14.02.2025.
[37]. Eclipse OpenJ9. Available at: https://eclipse.dev/openj9/, accessed 17.02.2025.
[38]. Kumar A.B.V. Supercharge Your Applications with GraalVM: Hands-on examples to optimize and extend your code using GraalVM's high performance and polyglot capabilities, Packt Publishing Ltd, 2021.
[39]. Introduction to GraalVM. Available at: https://www.graalvm.org/latest/introduction/, accessed 19.02.2025.
[40]. SpiderMonkey Bytecode Descriptions. Available at: https://udn.realityripple.com/docs/Mozilla/Projects/SpiderMonkey/Internals/Bytecode, accessed 17.02.2025.

[41]. SpiderMonkey. Available at: https://firefox-source-docs.mozilla.org/js/index.html, accessed 17.02.2025.
[42]. ECMA-335 -Common Language Infrastructure (CLI). Available at: https://ecma-international.org/publications-and-standards/standards/ecma-335/, accessed 14.02.2025.
[43]. Общие сведения о платформе .NET. Available at: https://learn.microsoft.com/ru-ru/dotnet/framework/get-started/overview, accessed 14.02.2025.
[44]. The Mono Runtime. Available at: https://www.mono-project.com/docs/advanced/runtime/, accessed 14.02.2025.
[45]. Understanding V8's Bytecode. Available at: https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775, accessed 18.02.2025.
[46]. V8 JavaScript engine. Available at: https://v8.dev/, accessed 18.02.2025.
[47]. Dalvik bytecode format. Available at: https://source.android.com/docs/core/runtime/dalvik-bytecode, accessed 17.02.2025.
[48]. Oh H.S. et al. Evaluation of Android Dalvik virtual machine. Proceedings of the 10th international workshop on java technologies for real-time and embedded systems, 2012, pp. 115-124.
[49]. Configure ART. Available at: https://source.android.com/docs/core/runtime/configure, accessed 17.02.2025.
[50]. Android runtime and Dalvik. Available at: https://source.android.com/docs/core/runtime, accessed 17.02.2025.
[51]. Introduction to Parrot. Available at: https://parrot.github.io/parrot-docs1/1.1.0/html/docs/intro.pod.html, accessed 17.02.2025.
[52]. A Parrot Overview. Available at: https://parrot.github.io/html/docs/overview.pod.html, accessed 17.02.2025.
[53]. Wang K. et al. Micro Virtual Machines: A Solid Foundation for Managed Language Implementation: Ph. D. Dissertation. Australian National University. DOI: 1885/147871, 2018.
[54]. The Mu Micro Virtual Machine. Available at: https://microvm.github.io/, accessed 14.02.2025.
[55]. Haas A. et al. Bringing the web up to speed with WebAssembly. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017, pp. 185-200.
[56]. Gurdeep Singh R., Scholliers C. WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, 2019, pp. 27-36.
[57]. Lauwaerts T., Singh R. G., Scholliers C. WARDuino: An embedded WebAssembly virtual machine. Journal of Computer Languages, 2024, vol. 79, pp. 101268.
[58]. Panda Bytecode Description. Available at: https://gitee.com/openharmony/arkcompiler_runtime_core/tree/master/static_core/isa, accessed 19.02.2025.
[59]. Panda Runtime. Available at: https://gitee.com/openharmony/arkcompiler_runtime_core/blob/master/static_core/docs/panda-runtime.md, accessed 19.02.2025.
[60]. JavaScript language overview. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_overview, accessed 18.02.2025.
[61]. Juneau J. et al. The definitive guide to Jython: Python for the Java platform. Apress, 2010.
[62]. Foord M., Muirhead C. IronPython in action. Manning Publications Co., 2009.
[63]. Операционная система Android. Available at: https://www.android.com/intl/ru_ru/what-is-android/, accessed 19.02.2025.
[64]. Irtoc tool. Available at: https://gitee.com/openharmony/arkcompiler_runtime_core/blob/master/static_core/docs/irtoc.md, accessed 18.03.2025.
[65]. Кулямин В.В. Методы верификации программного обеспечения, 2008, 111 с.
[66]. ГОСТ Р. 56939-2024. Защита информации. Разработка безопасного программного обеспечения. Общие требования. М.: Стандартинформ, 2024, 36 с.
[67]. Sirer E.G., Bershad B.N. Using production grammars in software testing. ACM SIGPLAN Notices, 1999, vol. 35, no. 1, pp. 1-13.
[68]. Информация о версиях JDK. Available at: https://www.java.com/releases/, accessed 24.12.2024.
[69]. The Microsoft Java Virtual Machine (MSJVM). Available at: https://en.wikipedia.org/wiki/Microsoft_Java_Virtual_Machine, accessed 24.12.2024.
[70]. Meyer J., Downing T. Java virtual machine. O'Reilly & Associates, Inc., 1997.

[71]. Jasmin is an assembler for the Java Virtual Machine. Available at: https://jasmin.sourceforge.net/, accessed 24.12.2024.
[72]. McKeeman W. M. Differential testing for software. Digital Technical Journal, 1998, vol. 10, no. 1, pp.100-107.
[73]. Sirer E. G., Bershad B. N. Testing Java virtual machines. Proc. Int. Conf. on Software Testing and Review, 1999.
[74]. Netscape Navigator. Available at: https://ru.wikipedia.org/wiki/Netscape_Navigator, accessed 24.12.2024.
[75]. Yoshikawa T., Shimura K., Ozawa T. Random program generator for Java JIT compiler test system. Third International Conference on Quality Software, 2003. Proceedings IEEE, 2003, pp. 20-23.
[76]. Calvagna A., Tramontana E. Automated conformance testing of Java virtual machines. 2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems. IEEE, 2013, pp.547-552.
[77]. NuSMV: a new symbolic model checker. Available at: https://nusmv.fbk.eu/, accessed 24.12.2024.
[78]. The Byte Code Engineering Library. Apache Commons BCEL. Available at: https://commons.apache.org/proper/commons-bcel/, accessed 24.12.2024.
[79]. Kyle S. et al. Application of domain-aware binary fuzzing to aid Android virtual machine testing. ACM SIGPLAN Notices, 2015, vol. 50, no. 7, pp. 121-132.
[80]. Chen Y. et al. Coverage-directed differential testing of JVM implementations. Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2016, pp. 85-99.
[81]. Jimple is the intermediate representation IR of Soot. Available at: https://soot-oss.github.io/SootUp/v1.1.2/jimple/, accessed 24.12.2024.
[82]. Soot – A framework for analyzing and transforming Java and Android applications. Available at: https://soot-oss.github.io/soot/, accessed 24.12.2024.
[83]. IBM SDK, Java Technology Edition, Version 8. Available at: https://www.ibm.com/support/pages/java-sdk-downloads-version-80, accessed 21.02.2025.
[84]. Guide to GNU gcj. Available at: https://gcc.gnu.org/onlinedocs/gcc-5.5.0/gcj/, accessed 21.02.2025.
[85]. Chen Y., Su T., Su Z. Deep differential testing of JVM implementations. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019, pp. 1257-1268.
[86]. DaCapo Benchmarks. Java. Available at: https://www.dacapobench.org/, accessed 24.12.2024.
[87]. OpenJDK. Available at: http://openjdk.java.net, accessed 24.12.2024.
[88]. Renouf C. The IBM J9 Java Virtual Machine for Java 6. Pro IBM WebSphere Application Server 7 Internals, 2009, pp. 15-34.
[89]. Polito G., Ducasse S., Tesone P. Interpreter-guided differential JIT compiler unit testing. Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2022, pp. 981-992.
[90]. Tikovsky J. R. Concolic testing of functional logic programs. International Workshop on Functional and Constraint Logic Programming. Cham: Springer International Publishing, 2017, pp. 169-186.
[91]. Polito G. et al. Cross-ISA testing of the Pharo VM: lessons learned while porting to ARMv8. Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, 2021, pp. 16-25.
[92]. Misse-Chanabier P. Testing a virtual machine developed in a simulation-based virtual machine generator: дис. Université de Lille, 2022.
[93]. Zhao Y. et al. History-Driven Test Program Synthesis for JVM Testing. In 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE). 1133–1144. https://doi.org/10.1145/3510003.3510059.
[94]. IBM Software Developers Kit (SDK). Available at: http://www.ibm.com/developerworks/java/jdk, accessed 24.12.2024.
[95]. Wu M. et al. JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). 56–68. DOI: 10.1109/ICSE48619.2023.00017.
[96]. Wu M. et al. SJFuzz: Seed and Mutator Scheduling for JVM Fuzzing. Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 1062-1074. https://doi.org/10.1145/3611643.3616277.
[97]. DragonWell от Alibaba. Available at: https://github.com/alibaba/dragonwell11, accessed 24.12.2024.
[98]. Zulu от Azul. Available at: http://www.azulsystems.com/products/zulu, accessed 24.12.2024.

[99]. Yang X. et al. Finding and understanding bugs in C compilers. Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, 2011, pp. 283-294.
[100]. Язык программирования C. Available at: https://www.open-std.org/JTC1/SC22/WG14/, accessed 28.02.2025.
[101]. Zhang Q., Sun C., Su Z. Skeletal program enumeration for rigorous compiler testing. Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation, 2017, pp. 347-361.
[102]. GCC, the GNU Compiler Collection. Available at: https://gcc.gnu.org/, accessed 04.03.2025.
[103]. Clang: a C language family frontend for LLVM. Available at: https://clang.llvm.org/, accessed 04.03.2025.
[104]. C Language Testsuites. Available at: https://gcc.gnu.org/onlinedocs/gcc-4.8.5/gccint/C-Tests.html, accessed 13.05.2025.
[105]. Zang Z. et al. Compiler testing using template java programs. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1-13.
[106]. Oracle JDK. Available at: https://www.oracle.com/ae/java/technologies/downloads/, accessed 25.02.2025.
[107]. Степанов Д.С. Исследование и разработка методов автоматического поиска ошибок в компиляторах языков программирования: диссертация на соискание уч. ст. к. т. н., Санкт-Петербург, 2024.
[108]. Blackburn S.M. et al. Rethinking Java Performance Analysis. Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, 2025, pp. 940-954.
[109]. Java Compatibility Kit 19 (JCK 19). Available at: https://www.oracle.com/bg/corporate/accessibility/templates/t2-12642.html, accessed 28.02.2025.
[110]. Shiv K. et al. SPECjvm2008 performance characterization. SPEC Benchmark Workshop, Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, pp. 17-35.
[111]. Howden W. E. Theoretical and empirical studies of program testing. IEEE Transactions on Software Engineering, 1978, no. 4, pp. 293-298.

Информация об авторах / Information about authors

Александр Сергеевич ПРОЦЕНКО – научный сотрудник отдела технологий программирования ИСП РАН. Область научных интересов: языковые виртуальные машины, микропроцессоры, архитектура системы команд, верификация и тестирование.

Alexander Sergeevich PROTSENKO is a researcher at the Software Engineering Department of ISP RAS. His research interests include language virtual machines, microprocessors, instruction set architecture, verification and testing.