DOI: 10.15514/ISPRAS-2025-37(1)-4

Применение формальных спецификаций системы команд для функционального тестирования языковых виртуальных машин

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. В настоящее время большой популярностью пользуются языки программирования, использующие в своей инфраструктуре языковые виртуальные машины (ВМ), что стимулирует развитие данной технологии. Разработка языковой ВМ – сложный процесс, в ходе которого могут вноситься ошибки в проектируемую систему. Для обеспечения качества реализации ВМ процесс разработки включает в себя этап тестирования. При тестировании необходимо ответить на два основных вопроса: как создавать тестовые программы и как проверять результат их исполнения. В статье представлен метод функционального тестирования языковых ВМ на основе формальных спецификаций системы команд. В работе описана реализация предложенного подхода. На основе документации ВМ специфицируется системы команд с помощью языка описания архитектуры. На основе спецификации системы команд строится исполнимая модель. Для автоматизации создания тестовых программ используются тестовые шаблоны – параметризованные описания тестовых программ. При создании тестовых шаблонов используется специальный предметно-ориентированный язык, позволяющий задавать различные техники генерации и использовать байт-код ВМ, полученный из формальных спецификаций. В представленном методе тестовые шаблоны могут быть описаны вручную, сгенерированы автоматически, в соответствии с целевым критерием, или получены от сторонних генераторов. На основе тестовых шаблонов и исполнимой модели генерируются тестовые программы на байт-коде, нацеленные на проверку определенной функциональности или свойств тестируемой системы. Байт-код является естественным языком для ВМ и позволяет воздействовать на всю ее функциональность. Тестовая программа транслируется в бинарную программу и исполняется на ВМ. Во время исполнения программы на ВМ собирается трасса исполнения. Для анализа трассы исполнения создается адаптер трасс. На основе исполнимой модели и адаптера трасс строится тестовый оракул. Оракул проверяет результаты тестирования путем сравнения трассы исполнения с результатами исполнения бинарной программы на исполнимой модели. Метод реализован в инструменте MicroTESK версии 2.6 и был использован для тестирования ВМ Ark.

Ключевые слова: тестирование на основе модели; языковая виртуальная машина; ВМ; система команд; архитектуры системы команд ISA; байт-код; формальные спецификации; тестирование; тестовая программа; тестовый оракул.

Для цитирования: Проценко А.С. Применение формальных спецификаций системы команд для функционального тестирования языковых виртуальных машин. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 65–86. DOI: 10.15514/ISPRAS–2025–37(1)–4.

Functional Testing of Language Virtual Machines Based on Formal ISA Specifications

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. Nowadays, programming languages that use language virtual machines (VMs) in their infrastructure are very popular, which stimulates the development of this technology. Developing a language VM is a complex process, during which errors can be introduced into the designed system. To ensure the quality of the VM implementation, the development process includes a testing stage. During testing, it is necessary to answer two main questions: how to create test programs and how to check the result of their execution. The article presents a method for functional testing of language VMs based on formal specifications of the instruction set architecture (ISA). The work describes the implementation of the proposed approach. Based on the VM documentation, the ISA is specified using the architecture description language. An executable model is built based on the ISA specification. Test templates, which are parameterized descriptions of test programs, are used to automate the creation of test programs. When creating test templates, a special domain-specific language is used, which allows you to specify various generation techniques and use the VM bytecode obtained from formal specifications. In the presented method, test templates can be described manually, generated automatically in accordance with the target criterion, or obtained from third-party generators. Based on the test templates and the executable model, test programs are generated in bytecode aimed at checking a certain functionality or properties of the system under test. Bytecode is a natural language for the VM and allows you to affect all of its functionality. The test program is translated into a binary program and executed on the VM. During the program execution, an execution trace is collected on the VM. A trace adapter is created to analyze the execution trace. A test oracle is built based on the executable model and the trace adapter. The oracle checks the test results by comparing the execution trace with the results of executing the binary program on the executable model. The method is implemented in the MicroTESK tool version 2.6 and was used to test the Ark (Panda) VM.

Keywords: model-based testing; language virtual machine; VM; ISA; bytecode; formal specification; testing; test oracle.

For citation: Protsenko A.S. Functional testing of language virtual machines based on formal ISA specifications. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025. pp. 65-86 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-4.

1. Введение

Концепция виртуальной машины (ВМ) была предложена около 50 лет назад и продолжает быть популярной и активно используемой. В настоящее время существует несколько вариантов классификации ВМ [1-2]. При этом выделяют два основных вида ВМ: системные ВМ (system virtual machine) и процессовые ВМ (process virtual machine). Системные ВМ (VirtualBox, QEMU, Xen и др.) предназначены для виртуализации всей компьютерной системы. Процессовые виртуальные машины предназначены для запуска отдельного приложения. Среди процессовых ВМ можно выделить большую группу языковых ВМ (Dalvik VM, Python VM, Java VM и др.), которые могут быть частью инфраструктуры ВМ для высокоуровневых языков программирования (High-Level Language VM).

В работе рассматриваются языковые ВМ, использующие систему команд, называемую байт-кодом (Java bytecode, Python bytecode, ARK Bytecode и др.). Концепция архитектуры системы команд (ISA) была описана впервые в работе [3] для системы команд микропроцессоров. Систему команд ВМ иногда называют виртуальной системой команд (virtual ISA, V-ISA).

Разработка языковой виртуальной машины – сложный процесс, в ходе которого могут вносится ошибки в проектируемую систему. Для обеспечения качества реализации ВМ в процесс разработки можно включить функциональное тестирование. Такое тестирование

обычно осуществляют при помощи: исполнения тестовых программ на тестируемой системе и проверки полученных результатов. Основными проблемами, рассматриваемыми в работе, являются: создание тестовых программ и проверка результата их исполнения на целевом устройстве.

Тестовая программа представляет собой тестовые воздействия, нацеленные на проверку определенной функциональности и/или аспекта тестируемой системы. Для автоматизации создания тестовых программ могут использоваться подходы на основе тестовых шаблонов. Тестовый шаблон представляет собой параметризованное описание структуры и функциональности тестовой программы. При этом тестовые шаблоны для ВМ должны иметь возможность использования метаданных, которые описывают классы, поля, методы, интерфейсы, объекты и другие артефакты программы. При тестировании ВМ можно выделить следующие области и уровни тестирования: транслятор, интерпретатор, загрузчик классов, верификатор байт-кода, сборщик мусора, модель памяти, оптимизатор, Just-in-Time (JIT) и Ahead-of-Time (AOT) компиляторы.

Байт-код для ВМ обычно получается с помощью трансляции с языка высокого уровня (Java, Kotlin, ArkTS). Для удачных реализация ВМ могут быть создано несколько языков высокого уровня, имеющих свои особенности при трансляции в байт-код. Поэтому для доступа ко всей функциональности ВМ необходимо использовать естественный для нее язык, а именно байт-код ВМ.

Источником информации о байт-коде и архитектурных особенностях ВМ является документация архитектуры системы команд. Поэтому методы, использующие архитектуру системы команд, могут быть реконфигурируемы, то есть настроены на систему команд, с помощью ее спецификаций.

Для проверки результата тестирования ВМ нужен тестовый оракул. Впервые термин “тестовый оракул” (test oracle) был использован в 1978 году [4]. Механизм работы оракула для ВМ заключается в сравнении результата исполнения тестовой программы на тестируемой системе и эталонного результата, который может быть получен с помощью исполнения программы на эталонной системе. Получение эталонной системы является непростой задачей, для решения которой могут быть использованы формальные спецификации.

В данной статье представлен метод функционального тестирования языковых виртуальных машин на основе формальных спецификаций системы команд. Метод заключается в следующем.

Шаг 1. На основе документации ВМ специфицируем систему команд ВМ. Для этого необходимо спроектировать и реализовать хранилище метаданных для классов, полей классов, методов, обработчиков исключений, объектов и полей объектов; необходимо специфицировать алгоритмы работы с метаданными: изменение стека фреймов при вызове и возврате из метода, создание объекта и поля объекта, вызов и возврат из метода, работа с полями объекта и статическими полями класса, загрузка метаданных в память для классов, полей классов, методов, обработчиков исключений, генерация и выброс исключения и др.

Шаг 2. На основе формальных спецификаций получаем исполнимую модель (симулятор).

Шаг 3. Описываем или генерируем тестовые шаблоны. Для описания тестовых шаблонов используется специализированный предметно-ориентированный язык (DSL) способный использовать техники генерации, языковые конструкции системы команд ВМ и метаданные. Тестовые шаблоны можно разрабатывать вручную, создавать с использованием сторонних генераторов или автоматически генерировать на основе анализа спецификаций в соответствии с заданным критерием тестового покрытия.

Шаг 4. На основе тестовых шаблонов и исполнимой модели генерируем тестовые программы.

Шаг 5. Исполняем полученные тестовые программы на ВМ. Получаем бинарный образ программы (*.class, *.abc и пр.) и трассу исполнения.

Шаг 6. Для анализа трассы исполнения и извлечения событий в трассе описывается адаптер трасс.

Шаг 7. На основе исполнимой модели и адаптера трасса создается оракул.

Шаг 8. Используем оракул для проверки результатов выполнения тестовых программ на ВМ.

Метод не привязан к конкретной ВМ, применим для большинства указанных составляющих ВМ и позволяет создать инструментарий для тестирования ВМ на всех этапах ее разработки и эксплуатации. Схема метода приведена на рис. 1.

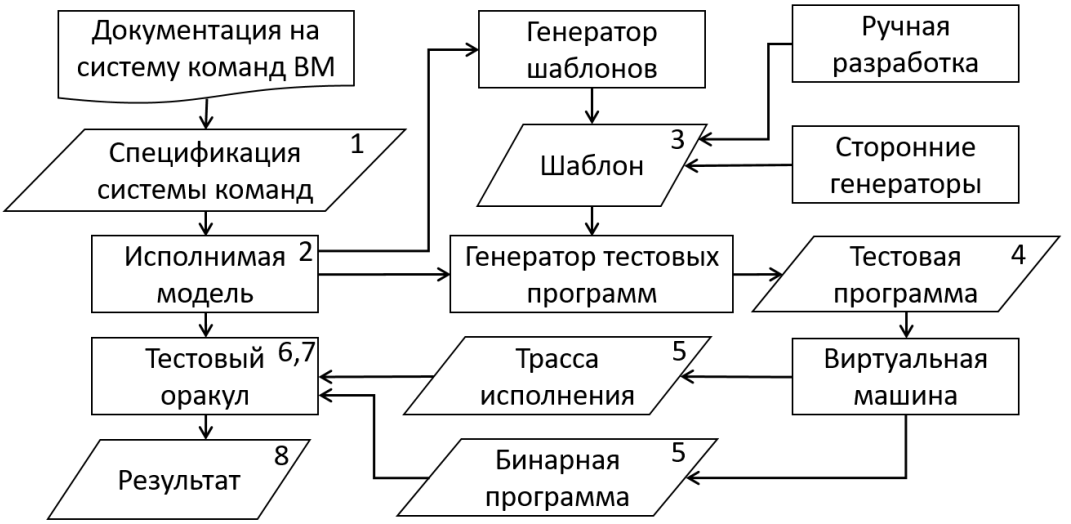

Рис. 1. Общая схема предлагаемого метода.
Fig. 1. General scheme of the proposed method.

Вклад статьи заключается в следующем:

1. Предложен метод функционального тестирования языковых ВМ на основе формальных спецификаций системы команд, впервые применяемый к тестированию ВМ.
2. Предложен метод спецификации систему команд ВМ на языке описания архитектуры.
3. Предложены методы создания тестовых программ на основе шаблонов и спецификации системы команд ВМ.
4. Предложен метод автоматизированного построения оракула на основе спецификации системы команд ВМ, для проверки результатов исполнения тестовых программ.
5. Описанные методы реализованы и применены к верификации системы команд ВМ Ark [5], где были найдены несоответствия в спецификации системы команд.

Остальная часть статьи организована следующим образом. В разделе 2 приводится обзор литературы по теме статьи. В разделе 3 описаны предлагаемые методы и их работа на примере ВМ Ark. В разделе 4 описаны полученные результаты. В разделе 5 приводится заключение.

2. Обзор литературы

Обзор литературы проводился в открытых источниках по работам, в которых описывались методы тестирование ВМ и методы похожие на предлагаемый в данной работе.

2.1 Похожая область, другие методы

В статье [6] описан метод тестирования, основанный на конколическом анализе путей исполнения инструкций в интерпретаторе и подготовке тестов, основанных на этой информации, для тестирования JIT-компилятора. Для проверки результатов сравниваются исполнения тестов интерпретатором в режимах с включенным и отключенным JIT-компилятором.

В статье [7] авторы предлагают подход тестирования Java Virtual Machine (JVM) [8], в основе которого лежат порождающие грамматики. В работе описывается предметно-ориентированный язык lava, применяемый для описания порождающих грамматик, с помощью которых генерируются тестовые программы.

В работах [9-14] описываются подходы для фаззинг тестирования ВМ. В качестве оракула используются ВМ разных версий и от разных компаний, реализующих одну спецификацию. Исходные программы для фаззинга в проектах берутся из различных открытых источников или генерируются на основе грамматических описаний.

В статье [15] проверяется корректность работы JIT оптимизации, путем проверки состояния стека ВМ при динамической JIT деоптимизации в случае нарушения охранных условий при выполнении оптимизированного кода.

В работе [16] проверяются ошибки, связанные с использованием некорректных типов операндов байт-код инструкций JVM. Авторы разделяют кодировки байт-код инструкций по группам в соответствии с типами операндов. Далее они комбинируют инструкции между собой и составляя пары. Полученные комбинации проверяют методом проверки моделей на формальной модели JVM описанной на NuSMV и получают информацию о корректности комбинаций инструкций. С помощью BCEL создают программы на байт-коде и проверяют их с помощью встроенного верификатора. Результат верификатора сравнивают с результатами от NuSMV, которые должны совпадать.

В диссертации [17] представлен метод тестирования ВМ, полученной с помощью среды генерации ВМ на основе моделирования. В качестве основы тестового набора используются тесты, созданные для проверки модели ВМ в среде моделирования. Этот набор расширяется тестами, полученными на их основе с помощью мутаций. Для проверки результата исполнения тестовых программ используется подход на основе дифференциального сравнения, в котором сравниваются результаты, полученные от сгенерированной ВМ на языке С и от ВМ из среды моделирования.

2.2 Другая область, похожие методы

Генераторы тестовых программ успешно используются для тестирования микропроцессоров. Различные подходы для построения тестовых программ были реализованы в инструментах: RIS (Random Instruction Sequence) от компании ARM [18], RAVEN (Random Architecture Verification Machine) от Obsidian Software (позже ARM) [19], Genesys-Pro используемый в IBM [20], прототип генератора MA2TG [21], генератор MicroTESK до версии 2.4 от ИСП РАН [22-25]. В данных работах для генерации тестовых программ обычно используются тестовые шаблоны, в которых есть возможность применения различных техник генерации и задания инструкций, которые необходимо добавить в тестовую программу. Некоторые инструменты могут быть настроены на целевую систему команд путем ее спецификации или с помощью описания семантики. Для проверки результатов тестирования обычно используют сравнение трасс, полученных от тестируемой RTL-модели и эталонной реализации (например, на языке С).

Основным отличием предлагаемого в данной работе подхода является: автоматическое построение оракула для проверки трасс исполнения тестовых программ, наличие механизмов работы с метаданными, автоматическое построение тестовых шаблонов. Оракул, в одном из

режимов работы, может проверять трассы с пропусками, что позволяет проверять трассы с использованием JIT и AOT компиляторов.

2.3 Похожие общие части

В работах [26-27] описано применение SMT-солверов для генерации данных для тестовых случаев. Для этого формальные предусловия переводятся в набор ограничений, разрешаемых SMT-солвером. Результаты выполнения таких тестов анализируются с помощью оракулов, основанных на формальных описаниях постусловий.

3. Описание метода и пример реализации

В разделе описаны детали предлагаемого метода функционального тестирования языковых ВМ и приведены примеры их реализации. В разделе представлены методы спецификации системы команд ВМ (раздел 3.1), создания тестовых программ на основе шаблонов и спецификации системы команд ВМ (раздел 3.2), автоматизированного построения оракула на основе спецификации системы команд ВМ (раздел 3.3).

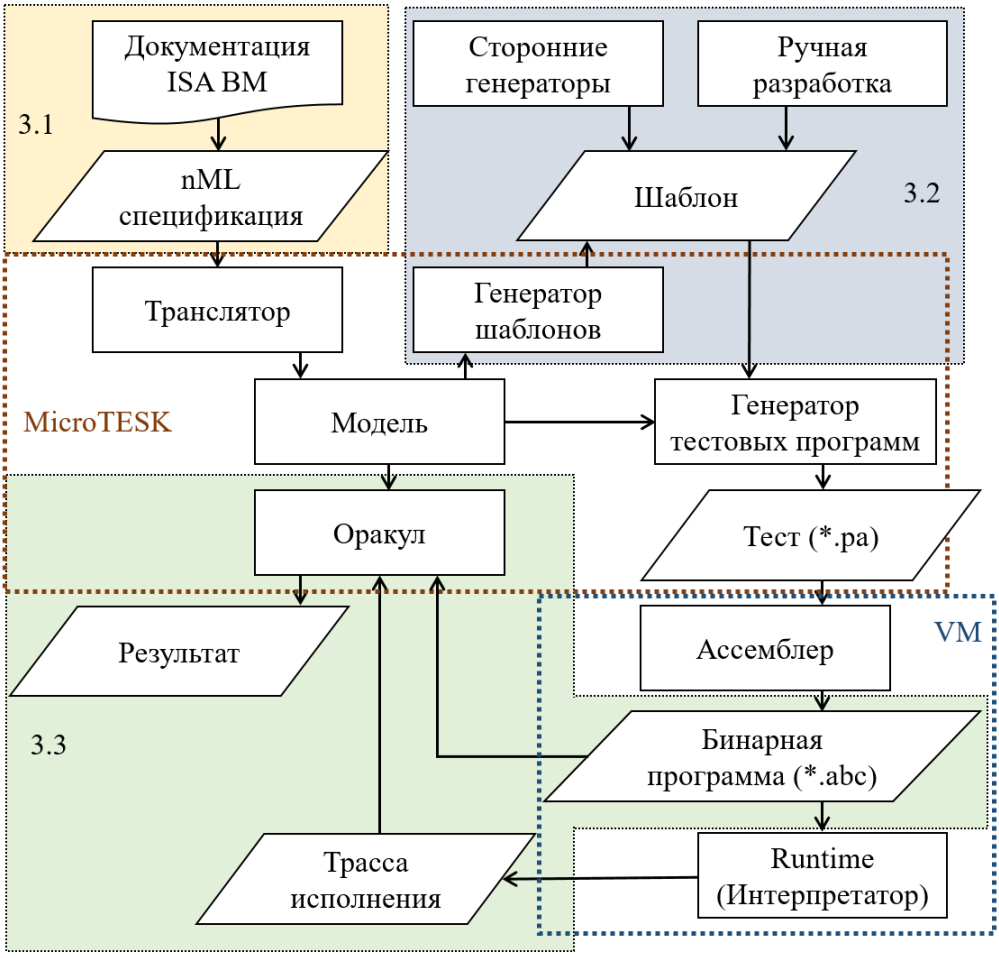

Рис. 2. Схема реализации метода функционального тестирования языковых виртуальных машин на основе формальных спецификаций системы команд.
Fig. 2. Scheme of implementation of the method for functional testing of language virtual machines based on formal ISA specifications.

На рис. 2 представлена расширенная схема основного метода с учетом деталей реализации. Метод реализован на основе инструмента с открытым исходным кодом MicroTESK [28]. Ранее инструмент MicroTESK использовался для тестирования микропроцессоров. Для возможности его применения для тестирования ВМ были внесены изменения, которые реализованы в версии 2.6. Описанный метод был применен к тестированию ВМ Ark [5], фрагменты реализации некоторых элементов из этого проекта будут использоваться в качестве примеров.

3.1 Спецификация системы команд на языке nML

3.1.1 Описание модели ВМ

Модель ВМ можно описать с помощью состояния и инструкций, изменяющих это состояние.

Состояние представляет собой память ВМ и включает в себя:

- глобальную статическую память, которая содержит метаданные, используемые для описания классов, методов, полей классов, обработки исключений и пр.;
- глобальную динамическую память, которая хранит данные артефактов, порождаемых во время работы ВМ и счетчик команд PC;
- локальную память, которая организована в виде стека, элементами которого являются фреймы, которые создаются при вызове метода ВМ и содержит данные необходимые для выполнения метода: регистры, стек операндов, стек локальных переменных и пр. После возвращения из метода фрейм уничтожается.

Инструкции описывают изменение состояния ВМ. Инструкции можно разбить на две группы:

- локальные инструкции, не зависящие от метаданных и не влияющие на глобальную память. Такие инструкции чаще всего представлены арифметическими инструкциями, инструкциями ветвления и инструкциями присваивания;
- глобальные инструкции, использующие метаданные и взаимодействующие с глобальной памятью, как статической, так и динамической. Такие инструкции создают экземпляры объектов, вызывают методы и осуществляют запись в поля объектов.

Инструкции производят операции над входными данными. Входные данные передаются в инструкцию через параметры. Параметры инструкции могут быть:

- явными (прямыми) – являющимися частью сигнатуры, в этом случае эти параметры являются и частью кодировки инструкции;
- не явными (косвенными) – использующие для передачи данных память, специальные регистры или стек. Такие параметры не являются частью кодировки инструкции.

Частью исполнения некоторых инструкций является генерация и выброс исключений. Если инструкция выбрасывает исключение, то она становится источником исключения. Исключение представляет собой экземпляр специального класса, содержащего описания исключительной ситуации в методе или инструкции. Для выброса исключения в коде метода используется специальная инструкция (обычно имеющая имя throw). Такая инструкция принимает созданный ранее экземпляр класса исключения как параметр. При выбросе исключения во время исполнения инструкции, например, инструкции деления (div), экземпляр класса исключения создается автоматически перед выбросом исключения.

Для обработки исключения используются try/catch блоки. Описание try/catch блоков содержит идентификатор метода, которому принадлежит блок обработки исключений, диапазон адресов метода, в котором перехватывается исключение, идентификатор класса

исключения для обработки и адрес обработчика исключения. После выброса исключения в блоке try в диапазоне указанных адресов, проверяется его соответствие типу перехватываемого исключения, при соответствии типа, управление передается по адресу обработчика исключения. Для одного диапазона адресов может быть описано несколько try/catch блоков. Если ни один из существующих try/catch блоков для данного диапазона адресов не смог обработать исключение, то метод завершает работу, а инструкция, вызвавшая данный метод, становится новым источником созданного исключения. Экземпляр исключения будет обработан try/catch блоками метода или программа завершится с ошибкой, описание которой будет взято из экземпляра исключения.

3.1.2 Метод спецификации системы команд ВМ

В качестве входной информации используется документация с описанием системы команд и архитектуры целевой ВМ.

Метод спецификации системы команд ВМ состоит из следующих шагов:

Шаг 1. На основе документации ВМ проектируется и специфицируется структура хранилища метаданных для классов, предков классов, методов классов, параметров методов классов, полей классов, объектов, полей объектов, обработчиков исключений. Для различных ВМ структура хранилища метаданных может различаться.

Шаг 2. Реализуются примитивы, необходимые для спецификации операций по работе с метаданными: генерация идентификаторов классов, методов, полей классов, объектов, полей объектов, обработчиков исключений; выделение памяти для метаданных классов, методов, полей классов, объектов, полей объектов, обработчиков исключений; завершение работы программы.

Шаг 3. Специфицируется основные операции для добавления, модификации и чтения метаданных: изменение стека фреймов при вызове и возврате из метода; создание объекта и поля объекта; вызов метода и возврат из метода; работа с полями объекта и статическими полями класса; загрузка метаданных в память для классов, полей классов, методов, обработчиков исключений; генерация и выброс исключения. Для спецификации операций используются определенные ранее примитивы и хранилище метаданных.

Шаг 4. Специфицируется локальные инструкции системы команд ВМ.

Шаг 5. Специфицируется глобальные инструкции системы команд ВМ. Используются операции для работы с метаданными, определенные ранее.

3.1.3 Пример nML спецификаций

Для спецификации состояния и инструкций ВМ был использован диалект языка описания архитектуры nML [29-31] разработанный в ИСП РАН. Ниже приведены примеры nML кода для различных элементов спецификации регистровой ВМ Ark.

Объявление общих регистров:

```
reg R [RF_SIZE, i64]
```

Объявление масок для регистров:

```
reg R_MASK [RF_SIZE, u64]
```

Объявление хранилища для типов хранимых данных.

```
var R_TYPE[RF_SIZE, u64]
```

Маски регистров показывают значащие биты, хранящиеся в регистре. Это необходимо для корректной работы оракула, так как регистры могут содержать значения разного типа, при этом часть битов регистров может быть не определена, и при сравнении оракулам эти значения в трассе и модели могут расходится. Константа `RF_SIZE` содержит значение

количества создаваемых регистров и масок, типы `i64` и `u64` соответствуют знаковому и беззнаковому 64 битному числу.

Для доступа к регистрам используются режимы доступа, их спецификация на nML представлена ниже:

```
mode V4 (i: card(4)) = R[i + REGISTER_SIZE*FC]
  init   = { register_mode = V4_MODE; }
  syntax = format("v%d", i)
  image  = format("%4s", i)
  action = { r_index = coerce(u16, i); }

mode V8 (i: card(8)) = R[i + REGISTER_SIZE*FC]
  init   = { register_mode = V8_MODE; }
  syntax = format("v%d", i)
  image  = format("%8s", i)
  action = { r_index = coerce(u16, i); }
```

Для работы с фреймами вводится специальная переменная FC. Когда создается новый фрейм, переменная FC увеличивается на 1, когда происходит возврат из фрейма, переменная уменьшается на 1. Переменная FC объявлена через ключевое слово `reg`, так как на текущий момент специального вида глобальных переменных не введено в nML. Объявление переменной FC на nML представлено ниже:

```
reg FC[FRAME_TYPE]
```

Для отслеживания текущего адреса инструкции используется счетчика инструкций (PC):

```
reg PC [i64]
```

Ниже приведено объявление памяти на nML для BM Ark:

```
shared mem MEM [MEM_SIZE, MEMORY_TYPE]
mem VM_MEM[VM_MEM_SIZE, MEMORY_TYPE]
```

Память `MEM` используется для хранения инструкций, и именно ее адреса ячеек хранит в себе `PC`. Память `VM_MEM` используется для хранения метаданных и экземпляров объектов.

Спецификация семантики инструкции виртуальной машины представляет собой процедуру и описывается на императивном языке, с помощью следующих элементов:

- Оператора присваивания;
- Управляющих конструкций;
- Рекурсивного поиска с заданными условиями;
- Составного оператора.

Управляющие конструкции представлены оператором с условием (if-then-else) и оператором возбуждения исключения. Управляющие конструкции и оператор присваивания используют набор битовых операции. Рекурсивный поиск с заданными условиями используется для работы с хранилищем метаданных. Составной оператор состоит из любого количества и любых комбинаций операторов присваивания, управляющих конструкций и рекурсивного поиска. Необходимыми типами данных для описания являются: битовые вектора произвольной длины и массивы. Для удобства описания также используется следующие типы: числа с плавающей точкой, списки и структуры. Стоит отметить, что вспомогательные типы можно смоделировать с помощью битовых векторов и массивов.

В описаниях инструкций управляющие регистры представлены глобальными переменными.

Фреймы представлены списком структур, содержащих атрибуты переменных и хранимые значения.

Ниже приведен пример спецификации локальной инструкции “div” на языке nML:

```
op div (vs1: V4, vs2: V4)
  init   = {image_size = OP_V4_V4_SIZE;}
  syntax = format("div %s, %s", vs1.syntax, vs2.syntax)
  image  = format("%s", op_v4_v4(DIV_V4_V4_OPCODE, vs1, vs2).image)
  action = {
    check_register_i32(vs1).action;
    check_register_i32(vs2).action;

    if (vs2 == 0) then
      add_exception(ArithmeticException).action;
    endif;

    if (vs1 == INT32_MIN && vs2 == -1) then
      acc_reg = coerce(i32, INT32_MIN);
    else
      acc_reg = vs1 / vs2;
    endif;

    set_acc_type_i32().action;
    set_acc_mask32().action;
  }
```

На листинге выше семантика инструкции **div** описана в атрибуте “`action`”, синтаксис описан в атрибуте “`syntax`”, в атрибуте “`image`” описана кодировка данной инструкции.

При возникновении исключения внутри инструкции, его обработка начинается с вызова метода `add_exception`, который передает управление конструктору класса исключения, где создается экземпляр исключения. При возбуждении исключения с помощью специальной инструкции `throw` на вход ей передается идентификатор экземпляра класса исключения. После выброса исключения начинается проверка, которая должна определить: находится ли инструкция, вызвавшая исключение, в try блоке. Если инструкция содержится в try блоке и исключение может быть обработано, осуществляется переход в catch блок, иначе происходит завершение исполнения метода и происходит проверка наличия подходящих try/catch блоков для инструкции, вызвавшей метод. Так будет происходить пока исключение не будет перехвачено или пока не будут завершены все методы.

Пример nML спецификации обработки выброса исключения представлен ниже. Приводится код внутренней операции `throw_recursion`, которая проверяет существование обработчика исключения для текущего метода, на адрес которого можно перейти.

```
internal op throw_recursion (class_id: MEMORY_TYPE)
  action = {
    this_method_id = frame_method_stack[FC];
    try_address   =   find_try_catch(TC_INDEX,   coerce(MEMORY_TYPE,
this_method_id), op_class_id, coerce(MEMORY_TYPE, CURRENT_PC));
    memory_get_element(try_address, EXISTENCE_FLAG).action;
    if (return_memory_element != 0) then
      // Найден подходящий обработчик исключений
      memory_get_element(try_address, HANDLER_PC_INDEX).action;
      temp1d1mt = return_memory_element;
```

```
        // Переход на catch блок
        PC = coerce(i64, temp1d1mt);
      else // Не найден обработчик исключений
        if (FC_VAR != 0) then // Не последний фрейм
          return_().action;
          CURRENT_PC = PC;
          // Флаг обозначающий повторный вызов этого метода
          return3mt = 1;
        else // Последний фрейм
          terminate_program = EXCEPTION_TERMINATE;
          if (main_terminate_address != 0) then
            PC = main_terminate_address;
          endif;
        endif;
      endif;
    }
```

В приведенном фрагменте кода операция `find_try_catch(...)` осуществляет рекурсивный поиск в хранилище метаданных обработчиков исключений с заданными параметрами. Листинг операции приведен ниже:

```
function find_try_catch(index: MEMORY_TYPE, value: MEMORY_TYPE,
                        id_class: MEMORY_TYPE, this_pc: MEMORY_TYPE)
                                : MEMORY_TYPE =
  if      VM_MEM[index + EXISTENCE_FLAG] != 0
    && !( VM_MEM[index + METHOD_ID_INDEX] == value
    && (VM_MEM[index + CATCH_CLASS_INDEX] == id_class ||
        VM_MEM[index + CATCH_CLASS_INDEX] == 0)
    && (VM_MEM[index + START_PC_INDEX] <= this_pc) &&
       (VM_MEM[index +   END_PC_INDEX] >= this_pc))
  then find_try_catch(index + TC_STR_SIZE, value, id_class, this_pc)
  else index
  endif
```

Операция `memory_get_element()` осуществляет чтение данных из памяти по заданному адресу и смещению. Листинг операции приведен ниже:

```
internal op memory_get_element(address: MEMORY_TYPE,
                               element: MEMORY_TYPE) action = {
  return_memory_element = VM_MEM[address + element];
}
```

На текущий момент спецификация системы команд на языке nML имеет ряд ограничений, связанных с ограничениями языка. Например, в текущей реализации глубина фреймов не может превышать 8.

3.2 Генерация тестовых программ

В предлагаемом методе функционального тестирования языковых виртуальных машин на основе формальных спецификаций системы команд тестовые программы генерируются на основе тестовых шаблонов и исполнимой модели. Исполнимая модель генерируется на основе спецификации системы команд из раздела 3.1.

Для создания тестовых шаблонов в предложенном методе используются следующие подходы:

- Ручная разработка тестовых шаблонов.
- Автоматическая генерация тестовых шаблонов.
- Использование сторонних генераторов.

В реализации метода для описания тестовых шаблонов используется DSL Ruby. Применение такого подхода обусловлено несколькими преимуществами [31], а именно: позволяет использовать байт-код специфицированной архитектуры, при описании методов; позволяет использовать встроенные техники генерации и генераторов данных; позволяет использовать возможности языка Ruby.

Спецификации системы команд были расширены специальными псевдо (`pseudo`) операциями для возможности загрузки метаданных, описывающих классы, поля классов, методы, обработчики исключений. Данные операции также доступны для использования в шаблонах.

Для представления и работы с метаданными в шаблоны были добавлены специальные структуры для классов, полей классов, методов и try/catch блоков. Пример такой структуры для поля класса приведен на листинге ниже:

```
Field = Struct.new(:id, :type, :class_id, :access_flag, :name)
```

При описании метода в шаблоне, помимо его описания с помощью соответствующей структуры, создается блок кода, содержащий байт-код метода. Адрес начала блока кода хранится в структуре описывающий этот метод.

Приведем пример описания простого класса:

```
def initialize_x_class()
  @@class_00 = Class.new get_new_class_id(), C_M_COUNT, C_F_COUNT
  @@f_00 = Field.new get_new_field_id(), FIELD_TYPE_I32,
      @@class_00.id, ACC_PUBLIC + ACC_STATIC, ""
  @@m_00 = Method.new get_new_method_id(), @@class_00.id,
      ACC_PUBLIC + ACC_STATIC, "", M_P_SHIFT, M_P_COUNT, M_ADDRESS
end
```

Такое описание позволяет создавать классы с нужным количеством полей определенных типов и режимов доступа, задавать контракты методов и описывать try/catch блоки соответствующих методов. Такое описание метаданных используется во всех способах создания тестовых шаблонов.

3.2.1 Ручная разработка тестовых шаблонов

Ручная разработка тестовых шаблонов применяется для начальной настройки базовых шаблонов и для описания шаблонов, которые сложно получить другими способами, отражающие специфику системы. В качестве иллюстрации начальной настройки приведем объявления регистров:

```
# Defines alias methods for V8 registers
(0..255).each do |i|
  define_method "v8_#{i}" do |&contents| V8(i, &contents) end
end
```

Описание препаратора для регистра V8:

```
preparator(:target => 'V8', :mask => "XXXX") {
  movi_v8_imm16 target, value(0, 15)
}
```

Препаратор описывает как загрузить данные в регистр ВМ, это используется, если регистр не был ранее инициализирован.

За счет использования встроенных генераторов и ранее описанных или сгенерированных элементов тестовых шаблонов данный подход становится менее трудозатратным и более удобным по сравнению с обычной ручной разработкой тестовых программ.

3.2.2 Автоматическая генерация тестовых шаблонов

Автоматическая генерация тестовых шаблонов позволяет получить шаблоны для создание тестовых программ, соответствующих определенным критериям тестового покрытия [32]. Метод автоматической генерации тестового шаблона на основе спецификации системы команд состоит из следующих шагов:

Шаг 1. Создаем абстрактный шаблон, описывающий структуру генерируемого тестового шаблона. В качестве элементов абстрактного шаблона могут использоваться наборы инструкций и генераторы данных.

Шаг 2. Получаем набор инструкций системы команд, соответствующие определенным критериям, из спецификаций системы команд.

Шаг 3. Определяем специализированные генераторы данных.

Шаг 4. На основе абстрактного шаблона, генераторов данных и набора инструкций генерируем тестовые шаблоны.

В качестве примера абстрактного шаблона рассмотрим структуру, состоящую из блока тестового воздействия и блока подготовки данных. Блок тестового воздействия состоит из одной инструкции из системы команд; в блоке подготовки данных используются препараторы, инициализирующие входные параметры инструкции данными, подготавливаемыми генератором данных в соответствии с заданным критерием тестового покрытия.

Рассмотрим в качестве критерия тестового покрытия, покрытие всех достижимых путей в графе потока управления инструкций. Для достижения этого тестового покрытия можно использовать специальный генератор данных, нацеленный на покрытие всех достижимых путей исполнения инструкции. Этот генератор реализован в инструменте MicroTESK и его можно использовать в тестовом шаблоне следующим образом:

```
sequence {
  div v4(_), v4(_) do testdata('all_paths') end
}.run
```

Генератор данных “all_paths”, нацеленный на покрытие всех достижимых путей исполнения инструкции, устроен следующим образом. На основе спецификации системы команд на языке nML строится внутреннее представление описываемых инструкций. Для каждой инструкции строится граф потока управления и соответствующая SSA-форма. Для каждого пути в графе на основе SSA-формы строится условие прохождения по данному пути: логическое выражение, использующее параметры процедуры и глобальное состояние модели. Затем проводится трансляция полученных условий в язык ограничений SMT-LIB. Для этого требуется представить глобальное состояние модели и непосредственно условия в терминах SMT-LIB. Далее проводится построение ограничений для SMT-решателя, с помощью которых можно сгенерировать входные данные для заданного условия пути. Подробнее про построение SMT-LIB формул можно прочитать в работе [33].

Стоит отметить, что на текущий момент реализация генератора “all_paths” не отслеживает зависимости по данным, считываемым из памяти, из-за чего для инструкций, работающих с метаданными, часть тестовых шаблонов была описана в полуавтоматическом режиме. Но это ограничение реализации, а не метода.

3.2.3 Использование сторонних генераторов

Использование сторонних генераторов позволяет получать тестовые шаблоны от других специализированных инструментов. Для этого можно применить метод создания архитектурно независимых тестовых шаблонов для виртуальных машин и микропроцессоров [34]. Метод состоит из следующих шагов:

Шаг 1. Формируем набор базовых конструкций, на основе анализа сценария тестового шаблона и архитектурно зависимых элементов, которые в нем используются. Базовые конструкции используются для замены архитектурно зависимых элементов.

Шаг 2. Создаем архитектурно независимый шаблон, используя сторонний генератор для получения тестового сценария, использующего базовых конструкций.

Шаг 3. Описываем реализации базовых конструкций. Реализация описывается с использованием системы команд, информация о которой может быть получена из спецификации системы команд. При этом реализация конструкций может быть описана несколькими способами, в зависимости от богатства системы команд. Для выбора используемых реализаций могут быть использованы различные стратегии перебора, например, случайный выбор или полный перебор.

В данной работе при реализации метода были выделены следующие базовые конструкции: присваивания, сложения, цикла с параметрами, условного оператора, операции завершения выполнения цикла. В качестве примера можно привести конструкцию условного оператора:

```
def if_condition_op(b, if_condition, if_label)
  if if_condition == JNE then
    jne_v8_imm16 V8(b), if_label
  elsif if_condition == JEQ then
…
    jne_v8_imm16 V8(b), if_label
  end
end
```

Для генерации архитектурно независимых шаблонов использовался внешний генератор для проверки оптимизаций, реализованных в ВМ. Принцип работы этого генератора описан в работе [35].

3.3 Проверка результата исполнения тестовой программы

После исполнения тестовой программы на ВМ, необходимо проверить полученный результат. В данной работе рассматривается два метода проверки результатов тестовой программы: использование встроенных самопроверок (self-checks) в тестовых программах и проверка трассы исполнения тестовой программы с помощью оракула. Для обоих методов используется симулятор, который строится по формальным спецификациям системы команд ВМ.

Встроенные самопроверки – это код на языке тестируемой ВМ, который сравнивает состояние тестируемой системы с состоянием модели, вычисленным во время генерации тестовой программы. Код для самопроверок описывается заранее и называется компаратором (comparator). Компараторы вставляются в тестовую программу в места, где необходимо проверить значения памяти. Эталонное значение для сравнения берется из симулятора. Этот метод хорошо подходит для случаев, когда отсутствует трасса или способы ее проверить. Недостатком является дополнительный код, вносимый в тестовые программы, отсутствие точной локализации места ошибки и отсутствие проверки синтаксиса и кодировок инструкций.

Оракул – это инструмент, который проверяет (объясняет) трассу исполнения тестовой программы путем сравнения ее событий с эталонными событиями, получаемыми при

исполнении тестовой программы на симуляторе. События трассы исполнения – это события, возникшие в процессе исполнения теста.

3.3.1 Метод автоматизированного построения оракула

Предлагаемый метод основан на использовании спецификации системы команд для получения декодера инструкций и исполнимой модели. Метод автоматизированного построения оракула состоит из следующих шагов:

Шаг 1. Формулируем список событий трассы исполнения, проверка которых должна осуществляться оракулом. Типами таких событий могут быть исполнение инструкции и вывод информации о текущем состоянии (память, регистры, стек).

Шаг 2. Описываем регулярные выражения для распознавания событий трассы.

Шаг 3. На основе регулярных выражений конструируется адаптер трасс. Результатом работы адаптера трасс является список событий трассы в одном из форматов для обмена данными, например, в формате *.JSON.

Шаг 4. Создаем инструмент (парсер) для извлечения метаданных и списка инструкций метода из исполнимого файла тестовой программы.

Шаг 5. На основе спецификации системы команд получаем декодер инструкций.

Шаг 6. На основе спецификации системы команд получаем исполнимую модель.

Шаг 7. Создаем механизм загрузки метаданных и декодированных инструкций метода в память модели.

Шаг 8. Связываем события трассы с событиями в исполнимой модели.

Шаг 9. Формулируем критерии сравнения событий.

Шаг 10. Реализуем алгоритмы симуляции тестовой программы и проверки событий. Например, может быть использовано несколько стратегий: для проверки трасс с использованием JIT оптимизаций и без них.

Полученный таким способом оракул принимает на вход трассу исполнения и исполнимый файл тестовой программы. Результатом работы оракула должен быть вердикт о соответствии трассы и тестовой программы. В зависимости от реализации может выводиться дополнительная информация, например, об различиях в событиях трассы и симулятора, или об отсутствующих инструкций в трассе.

3.3.2 Схема работы оракула

Опишем подробнее схему проверки трассы исполнения тестовой программы с помощью оракула:

Шаг 1. Средствами ВМ проверяем тестовую программу на корректность с помощью верификатора и получаем бинарную программу. Бинарная программа является исполнимым файлом и может быть выполнена на ВМ.

Шаг 2. Исполняем бинарную тестовую программу на ВМ и получаем трассу исполнения.

Шаг 3. Получаем метаинформацию и байт-код методов. Для этого подаем на вход парсеру бинарную программу, полученную ранее.

Шаг 4. Собираем события трассы. Для этого используем адаптер трасс, на выходе получаем список событий в формате JSON.

Шаг 5. Получаем результат проверки трассы тестовой программы. На вход оракул принимает: события трассы исполнения в JSON формате, метаданные программы и байт-код методов. На основе этих данных оракул выносит вердикт о соответствии или несоответствии полученной трассы поданной тестовой программе.

Данная схема представлена на рис. 3. Детали реализации алгоритма работы оракула описаны в следующем подразделе.

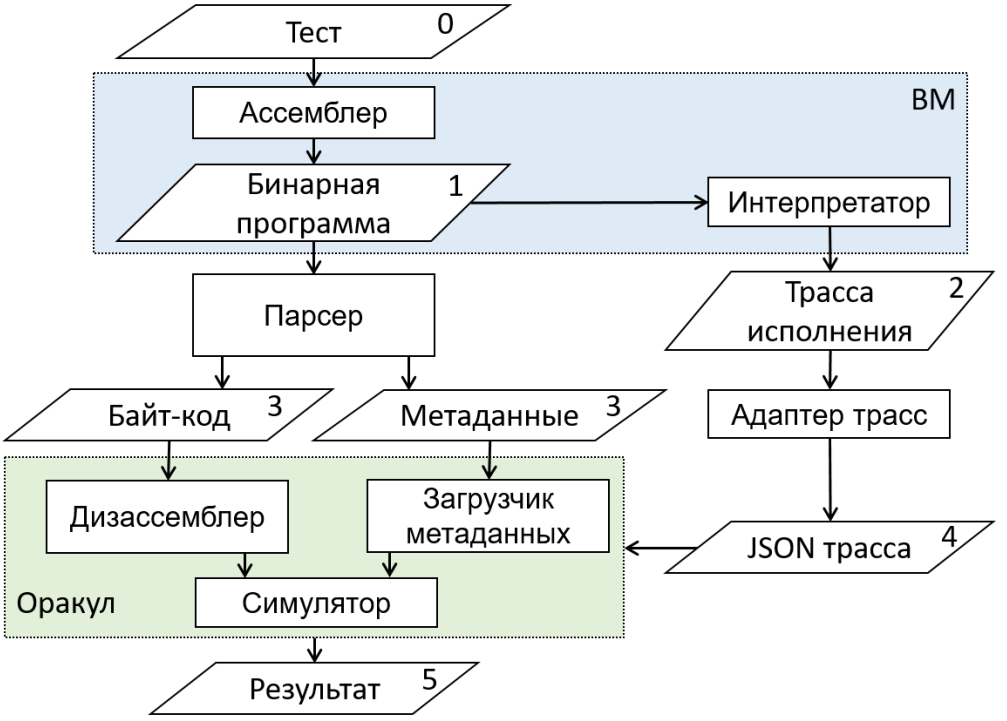

Рис. 3. Схема проверки трассы с помощью оракула.
Fig. 3. Scheme for checking the test program trace using oracle.

3.3.3 Пример работы оракула

Основные элементы оракула: симулятор и дизассемблер строятся автоматически инструментом MicroTESK на основе формальных спецификаций на nML. С помощью дизассемблера распознается байт-код методов из бинарной программы. Симулятор используется для исполнения байт-кода. Загрузка метаданных осуществляется через специальный модуль инструмента, позволяющий загружать метаданные в память симулятора. Для симуляции программы оракулу необходима вся информация об используемых классах. Если в программе используются библиотечные классы, то информацию о них необходимо подготовить и загрузить в симулятор.

Для анализа трассы исполнения от ВМ применяется адаптер трасс. Ниже приведен пример фрагмента трассы irtoc интерпретатора ВМ Ark:

```
[TID 031b4e] D/interpreter:  pc: 0x7f30ca8bf173 ---> movi v0, 118
[TID 031b4e] D/interpreter:   acc.pri = (i64) 0 | (f32) 0
                                      | (f64) 0 | (hex) 0
[TID 031b4e] D/interpreter:   v0.pri = (i64) 118 | (f32) 1.65353e-43
                                     | (f64) 5.82997e-322 | (hex) 76
```

Для анализа событий адаптер трасс использует регулярные выражения, что позволяет его настраивать на различные виды трасс. С помощью адаптера трасс собираются события из трассы и представляются в специальном JSON формате, что позволяет сравнивать их с исполняемым в симуляторе байт-кодом и состоянием симулятора. Ниже приведен пример распознанных событий в формате JSON трассы: исполнение инструкции и вывод информации о текущем состоянии (памяти):

```
{"EventType": "INSTRUCTION", "Instruction": "movi v0, 118",
              "ThreadIdentifier": "031b4e", "PC": "7f30ca8bf173"},
{"EventType": "MEMORY", "ThreadIdentifier": "031b4e", "RegisterType":
              "pri", "RegisterValue": "0", "Register": "acc"},
{"EventType": "MEMORY", "ThreadIdentifier": "031b4e", "RegisterType":
              "pri", "RegisterValue": "76", "Register": "v0"}
```

Оракул читает событие из трассы: если это состояние, то он проверяет совпадает ли оно с состоянием в симуляторе; если это инструкция, то оракул сравнивает ее со следующей инструкцией в очереди и исполняет ее. Если оракул работает в режиме полного совпадения, то любое расхождение является признаком несоответствия трассы программе. Ниже приведен пример работы оракул для сравнения событий:

```
Compare Instruction: [SUCCESS],
                     (trace : model) -> 'movi v0, 118' : 'movi v0, 118'
[ProgramSimulator] PC: 0x64, call: movi v0, 118
Compare Registers: [SUCCESS], Mask: 0x0000000000000000,
                     (trace : model) -> 'acc' = 0x0 : 'ACC_G' = 0x0
Compare Registers: [SUCCESS], Mask: 0x00000000FFFFFFFF,
                     (trace : model) -> 'v0' = 0x76 : 'R[0]' = 0x76
```

Оракул в режиме с пропусками, в случае расхождения, начинает перебирать следующие инструкции и состояния для сравнения. Признаком успешности для него является нахождение соответствия всем события в трассе, при этом он выводит список инструкций, которых в трассе обнаружено не было. Такой режим оракула, в отличии от простого сравнения трасс, применим для проверки трасс, полученных после JIT- и AOT- компиляции, дающих трассы с пропусками.

4. Полученные результаты

Описываемый в работе метод функционального тестирования языковых ВМ на основе формальных спецификаций применялся для тестирования ВМ Ark. При тестировании ВМ были найдены несоответствия в спецификации системы команд и ошибки в реализации некоторых механизмов ВМ.

На момент написания статьи на языке nML было специфицировано 298 инструкций/байт-код команд ВМ Ark. Подробнее информация по группам инструкций представлена в табл. 1.

Табл. 1. Статистика по специфицированным инструкциям виртуальной машины Ark.
Table 1. Statistics for Ark VM specified instructions.

Группа инструкций	Кол-во	Спец.
Арифметические	131	131
Управляющие	35	35
Арифметические для чисел с плавающей точкой	34	34
Работы с регистрами	32	32
Работы с памятью/Системные	42	42
Динамические	6	6
Работы с массивами	18	18
Всего	298	298

Сгенерированный тестовый набор для проверки работы отдельных инструкций ВМ содержит 496 тестовых программ. Покрытие кода ‘cpp’ интерпретатора ВМ данным тестовым набором составило 85%. Измерение покрытия кода проводилось с помощью инструмента **gcov** [36]. Отчасти такой процент покрытия можно объяснить тем, что интерпретатор содержит код, для обработки ситуаций отсутствия в исполняемом файле классов, методов и пр., который покрыт не был.

С помощью данного тестового набора было найдено несколько ошибок. Отсутствовали реализации 2х инструкций, описанных в документации, и отсутствовала возможность корректного вызова еще 3х инструкций. Найдены мелкие ошибки в синтаксисе инструкций в трассе исполнения программы. С помощью этого же тестового набора было найдено расхождение поведения верификатора для однотипных инструкций, в которых присутствовала обработка ситуации, при подаче на вход null объекта. В одном случае верификатор не сообщал о проблемах, во втором случае сообщал об ошибке с сообщением о невозможности подачи на вход null объекта.

Были найдены ошибки с помощью тестовых программ, полученных из шаблонов, написанных вручную. Одна из таких ошибок связана с использованием try/catch блоков. Пример метода такой тестовой программы приведен ниже:

```
.function i64 class10.method0(i64 a0, i64 a1) {
  nop
try_start_instruction:
  nop
# Preparation
  ldai.64 0
# Stimulus
  not.64
try_end_instruction:

  nop
  ststatic.64 class10.field0
catch_instruction_block_begin:
  return.64

  .catchall try_start_instruction, try_end_instruction,
catch_instruction_block_begin
}
```

Такая программа успешно проходила проверку верификатором ВМ Ark, успешно исполнялась интерпретаторами ВМ и соответствовала предоставленной на тот момент документации. Но при попытке применения к методу JIT или AOT компиляторов выдавалась ошибка. Проблема продемонстрированного метода заключалась в том, что в качестве выхода из метода используется инструкция return.64 из блока обработки исключения. Стоит особенно отметить, что сложно представить, как такой тестовый пример можно получить с языка высокого уровня, что подтверждает полезность методов генерации тестовых программ на уровне байт-кода.

Еще одна интересная ошибка была найдена тестовой программой, полученной из шаблона, разработанного вручную, для проверки работы регистров с максимальными индексами в методе. Максимально возможное значение индекс регистра в методе ВМ Ark равно 65535 (2^16). Но передаваемые в метод параметры загружаются в регистры, следующие за максимальным используемом в методе. И при использовании в методе регистра с максимальным индексом 65535, передаваемые параметры загружались в регистры с индексами 65536, 65537, что являлось ошибкой.

5. Дальнейшие исследования

На основе опыта применения метода и полученных результатов можно сформулировать следующие идеи и направления для дальнейшего развития подхода и прототипа:

- Автоматизация создания тестовых наборов на основе генерации тестовых последовательностей из 2-х и более взаимодействующих инструкций. При тестировании микропроцессоров большое количество ошибок наблюдается при выполнении цепочек инструкций [22, 37-38] на тестируемом устройстве. Предлагаемый подход можно расширить генераторами и шаблонами, нацеленными на покрытие исполнения цепочек инструкций.
- Расширение набора автоматически генерируемых тестовых шаблонов и модели ошибок для них.
- Автоматизированное создание препараторов и компараторов для тестовых шаблонов. В подходе при описании шаблонов, некоторые части по-прежнему приходится описывать вручную. К таким частям можно отнести: препараторы, необходимые для инициализации локальной памяти (регистров) и компараторов, используемых при проверках состояния локальной памяти.
- Развитие языка nML. На текущий момент спецификация ISA BM Ark на nML имеет ряд ограничений, например, максимальное количество доступных фреймов ограничено 8.
- Расширение списка генераторов данных и структурных генераторов, используемых в инструменте MicroTESK.

6. Заключение

В данной статье предложен метод функционального тестирования языковых ВМ на основе формальных спецификаций. В работе было показана возможность написания спецификаций на языке описания архитектуры nML для системы команд ВМ. В работе показано применение тестовых шаблонов с использованием метаданных, которые позволяют создавать классы с заданными требованиями. В описанном подходе основным способом воздействия на ВМ являются программы на байт-коде, что позволяет воздействовать на всю функциональность ВМ. Было показано, как на основе формальных спецификаций можно получить тесты для покрытия всех достижимых путей исполнения инструкции. Был описан метод, позволяющий использовать сторонние генераторы для получения тестов, нацеленных на проверку корректности оптимизации и предложена концепция архитектурно независимых шаблонов. Для проверки результатов исполнения тестовых программ в подходе используется оракул, создаваемый на основе спецификаций системы команд. Оракул позволяет проверять обычные трассы и трассы с пропусками, полученные при использовании JIT- и AOT-компиляции. Данный подход был применен к тестированию ВМ Ark во время ее разработки. Было получено высокое (85%) покрытие кода интерпретатора, найдены ошибки в спецификации системы команд и в реализации ВМ. Метод показал свою полезность при функциональном тестировании ВМ и имеет перспективы развития.

Список литературы / References

[1]. Smith J., Nair R. Virtual machines: versatile platforms for systems and processes. – Elsevier, 2005.
[2]. Li X. F. Advanced design and implementation of virtual machines. – CRC Press, 2016.
[3]. Amdahl G. M., Blaauw G. A., Brooks F. P. Architecture of the IBM System/360 //IBM Journal of Research and Development. – 1964. – Т. 8. – №. 2. – С. 87-101.
[4]. Howden W. E. Theoretical and empirical studies of program testing //IEEE Transactions on Software Engineering. – 1978. – №. 4. – С. 293-298.

[5]. Open-source проект Ark (Panda) Runtime [Электронный ресурс]. – Режим доступа: https://gitee.com/openharmony/arkcompiler_runtime_core/tree/master/static_core/
[6]. 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. – С. 981-992.
[7]. Sirer E. G., Bershad B. N. Using production grammars in software testing //ACM SIGPLAN Notices. – 1999. – Т. 35. – №. 1. – С. 1-13.
[8]. Java Language and Virtual Machine Specifications [Электронный ресурс]. – Режим доступа: https://docs.oracle.com/javase/specs/
[9]. Sirer E. G., Bershad B. N. Testing Java virtual machines //Proc. Int. Conf. on Software Testing And Review. – 1999.
[10]. 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. – С. 85-99.
[11]. Chen Y., Su T., Su Z. Deep differential testing of JVM implementations //2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). – IEEE, 2019. – С. 1257-1268.
[12]. 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 [Электронный ресурс]. https://doi.org/10.1145/3510003.3510059
[13]. 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 [Электронный ресурс]. DOI: 10.1109/ICSE48619.2023.00017
[14]. 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. – С. 1062-1074.
[15]. Zang Z. et al. Compiler testing using template java programs //Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. – 2022. – С. 1-13.
[16]. Calvagna A., Tramontana E. Automated conformance testing of Java virtual machines //2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems. – IEEE, 2013. – С. 547- 552.
[17]. Misse-Chanabier P. Testing a virtual machine developed in a simulation-based virtual machine generator: дис. – Université de Lille, 2022.
[18]. Hrishikesh M.S., Rajagopalan M., Sriram S., Mantri R. System Validation at ARM — Enabling our Partners to Build Better Systems. White Paper. April 2016
[19]. Mahapatra R.N., Bhojwani P., Lee J., and Kim Y. Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No.43 Phase 3 Report. 2009. 43 P.
[20]. Adir A. et al. Genesys-pro: Innovations in test program generation for functional processor verification //IEEE Design & Test of Computers. – 2004. – Т. 21. – №. 2. – С. 84-93.
[21]. Li T. et al. MA/sup 2/TG: a functional test program generator for microprocessor verification //8th Euromicro Conference on Digital System Design (DSD'05). – IEEE, 2005. – С. 176-183.
[22]. Камкин А. С. Генерация тестовых программ для микропроцессоров //Труды Института системного программирования РАН. – 2008. – Т. 14. – №. 2. – С. 23-63.
[23]. Kamkin A. Combinatorial model-based test program generation for microprocessors //Preprint of ISPRAS. – 2009.
[24]. Chupilko M., Kamkin A., Protsenko A., Smolov S., Tatarnikov A. MicroTESK: Automated Architecture Validation Suite Generator for Microprocessors. DVCon Europe 2018.
[25]. Kamkin A., Tatarnikov A. MicroTESK: A Tool for Constrained Random Test Program Generation for Microprocessors //Perspectives of System Informatics: 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers 11. – Springer International Publishing, 2018. – С. 387-393.
[26]. Peña R. et al. SMT-Based Test-Case Generation and Validation for Programs with Complex Specifications //Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems: Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday. – Cham: Springer Nature Switzerland, 2023. – С. 188-205.
[27]. Khurshid S., Marinov D. TestEra: Specification-based testing of Java programs using SAT //Automated Software Engineering. – 2004. – Т. 11. – С. 403-434.
[28]. Open-source project MicroTESK [Электронный ресурс]. – Режим доступа: https://forge.ispras.ru/projects/microtesk

[29]. Freericks M. The nML machine description formalism. – Leiter der Fachbibliothek Informatik, Sekretariat FR 5-4, 1991.
[30]. Vishnoi S. K. Functional simulation using sim-nml //Master's thesis, Department of Computer Science and Engineering, IIT Kanpur. – 2006.
[31]. Татарников А.Д. Автоматизация конструирования генераторов тестовых программ для микропроцессоров на основе формальных спецификаций. Диссертация на соискание ученой степени к.т.н. Институт системного программирования РАН, Москва, 2017. 162 с.
[32]. Проценко А. С., Татарников А. Д. Автоматическая генерация тестовых шаблонов на основе спецификаций системы команд //Новые информационные технологии в исследовании сложных структур. – 2018. – С. 82-83.
[33]. Kamkin A., Khoroshilov A., Kotsynyak A., Putro P. Deductive binary code verification against source-code-level specifications //Tests and Proofs: 14th International Conference, TAP 2020, Held as Part of STAF 2020, Bergen, Norway, June 22–23, 2020, Proceedings 14. – Springer International Publishing, 2020. – С. 43 - 58.
[34]. Проценко А. С. Архитектурно независимые тестовые шаблоны для виртуальных машин и микропроцессоров. ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ, Том 30, № 11, 2024, С. 579–584. doi:10.17587/it.30.579-584
[35]. Zelenov S., Zelenova S. Model-based testing of optimizing compilers //International Workshop on Formal Approaches to Software Testing. – Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. – С. 365-377.
[36]. gcov – a Test Coverage Program [Электронный ресурс]. – Режим доступа: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[37]. TMS570LS31x/21x Series Microcontroller, Silicon Errata. Texas Instruments, 2013. https://www.ti.com/lit/er/spnz195g/spnz195g.pdf
[38]. dsPIC30F5011/5013 Rev A1/A2 Silicon Errata. Microchip, 2005. https://ww1.microchip.com/downloads/en/DeviceDoc/80210e.pdf

Информация об авторах / Information about authors

Александр Сергеевич ПРОЦЕНКО является научным сотрудником 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.