УДК 004.4'42, 004.4'418, 004.434
doi:10.31799/1684-8853-2024-4-24-39
EDN: RIUJGA

Научные статьи
Articles

Моделирование поведения функций стандартной библиотеки в задачах анализа программ

***В. М. Ицыксон*[а]**, *канд. техн. наук, профессор, orcid.org/0000-0003-0276-4517, itsykson@yandex.ru*
***М. П. Онищук*[б]**, *аспирант, orcid.org/0000-0001-5359-0161*
***В. В. Кечин*[а]**, *магистрант, orcid.org/0009-0006-0845-2740*
***Я. А. Алексеев*[в]**, *инженер-программист, orcid.org/0009-0002-6925-6541*
[а]*Университет ИТМО, Кронверкский пр., 49, Санкт-Петербург, 197101, РФ*
[б]*Санкт-Петербургский политехнический университет Петра Великого, Политехническая ул., 29, Санкт-Петербург, 195251, РФ*
[в]*ООО «Коулмэн Тех», Мира пр., 40, помещ. 3/1, Москва, 129090, РФ*

Введение: *статический анализ программных проектов, использующих внешние библиотеки, сопряжен с экспоненциальным ростом числа анализируемых трасс программы. Это приводит к необходимости искусственно ограничивать время анализа или использовать какие-либо упрощения, в результате чего ухудшаются полнота и точность анализа, снижается тестовое покрытие.* ***Цель:*** *разработать подход к моделированию поведения внешних библиотек на основе формальных спецификаций, аппроксимирующих поведение исходных функций, позволяющий упростить анализируемые проекты и снизить время анализа.* ***Методы:*** *моделирование функций стандартной библиотеки с помощью формальных спецификаций на языке LibSL и последующий автоматический синтез простых эквивалентов библиотечных функций на основе формальных описаний.* ***Результаты:*** *предложен подход к спецификации функций стандартной библиотеки языка Java на языке LibSL, заменяющий наиболее часто используемые библиотеки и их функции на формальные спецификации, повторяющие видимое извне поведение функций. Полученные аппроксимирующие описания транслируются в эффективные для анализа замены функций стандартной библиотеки и подставляются анализатору вместо исходных библиотечных функций. Предложенный подход реализован в виде набора спецификаций части стандартной библиотеки Java и транслирующих утилит. Проведенные эксперименты на части стандартной библиотеки языка Java показали применимость подхода: время анализа промышленных проектов сократилось в среднем на треть.* ***Практическая значимость:*** *разработанный подход позволяет значительно расширить область применения статического анализа в задачах обнаружения дефектов и генерации тестов, так как ввиду сокращения пространства состояний увеличивается точность и полнота обнаружения дефектов и в большинстве случаев обеспечивается более высокое покрытие программы сгенерированными тестами.*

Ключевые слова *— стандартная библиотека языка, формальные спецификации библиотек, моделирование поведения библиотеки, статический анализ программ, генерация тестов.*

Для цитирования: Ицыксон В. М., Онищук М. П., Кечин В. В., Алексеев Я. А. Моделирование поведения функций стандартной библиотеки в задачах анализа программ. *Информационно-управляющие системы*, 2024, № 4, с. 24–39. doi:10.31799/1684-8853-2024-4-24-39, EDN: RIUJGA

For citation: Itsykson V. M., Onischuck M. P., Kechin V. V., Alekseev Y. A. Modeling the behavior of standard library functions in program analysis. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2024, no. 4, pp. 24–39 (In Russian). doi:10.31799/1684-8853-2024-4-24-39, EDN: RIUJGA

Введение

В то время как объем программного обеспечения (ПО) увеличивается, сферы его применения расширяются, а количество критически важных областей жизни человека, зависящих от программного обеспечения, растет, проблема качества ПО встает все более остро. Ошибки в программном обеспечении все сильнее влияют на процессы жизнедеятельности человека. Компании-разработчики ПО уделяют все больше времени качеству разрабатываемого продукта. Одними из самых перспективных методов оценки и повышения качества ПО являются методы, основанные на статическом анализе. К ним относятся, например, методы статического анализа для обнаружения дефектов ПО и для автоматической генерации тестов. Первые находят имеющие в программном коде ошибки, вторые генерируют тесты, запуск которых позволяет обойти всевозможные пути исполнения программы и обнаружить ошибки.

Основными характеристиками методов поиска ошибок являются ресурсоемкость, полнота и точность обнаружения. Полнота обнаружения (recall) показывает долю найденных методом ошибок в программе по отношению ко всем ошибкам в программе. Точность обнаружения (precision) показывает долю истинных ошибок среди всех найденных анализатором ошибок.

Основными характеристиками методов генерации тестов являются ресурсоемкость, тестовое по-

крытие и качество тестов. Тестовое покрытие показывает долю покрытых тестами элементов программы по отношению ко всем элементам программы. В зависимости от типов элементов фактически могут использоваться разные виды покрытия: покрытие строк, операторов, ветвлений, условий и т. п. Под качеством тестов понимается несколько характеристик сгенерированного множества тестов, в том числе доля дублирующихся тестов, доля тестов, не расширяющих тестового покрытия, и т. д.

Главным вызовом для всех статических методов является анализ многокомпонентных проектов, активно использующих внешние библиотеки. В таких проектах объем анализируемого программного кода чрезвычайно большой, так как кроме анализа программ, созданных разработчиками, инструменты должны обрабатывать огромные массивы артефактов, связанных с используемыми внешними библиотеками: исходный код, байт-код или объектные модули. Анализ таких больших объемов данных сопряжен с рядом проблем.

В данной статье описывается разработанный авторами подход к решению проблемы анализа многокомпонентных проектов, основанный на моделировании поведения стандартных библиотек. Подход заключается в создании формальных спецификаций библиотек на языке LibSL, последующей трансляции построенных спецификаций в упрощенные эквиваленты функций стандартных библиотек и замене в анализируемом проекте реальных библиотечных функций на модельные в целях снижения пространства состояний анализа, и, как следствие, времени работы анализатора. Разработанный подход был апробирован на части стандартной библиотеки языка Java и показал существенное снижение времени анализа на индустриальных проектах.

Существующие подходы к моделированию библиотек

Проблема моделирования сторонних библиотек находит свое решение во многих инструментах статического анализа. Существует несколько основных подходов, реализуемых разными анализаторами:

— полный анализ программы вместе со всеми используемыми библиотеками;

— игнорирование при анализе внешних библиотек;

— встраивание в анализаторы моделей наиболее часто используемых функций стандартной библиотеки;

— замена библиотечных функций при анализе на примитивные модели, описанные с использованием предметно-ориентированного языка (Domain-Specific Language, DSL);

— замена библиотечных функций при анализе на сложные аппроксимации, написанные на целевом языке;

— замена библиотечных функций при анализе на аппроксимации, полученные из модельного представления функции.

При полном анализе многокомпонентных проектов инструменты сталкиваются с взрывом числа состояний, который, как следствие, приводит к катастрофическому увеличению времени анализа. Для сокращения времени работы инструментов анализа используются разные подходы, связанные с искусственным лимитированием глубины или времени анализа, что позволяет снизить ресурсоемкость, но приводит к ухудшению параметров полноты и точности обнаружения ошибок и снижению покрытия и ухудшению качества тестов при генерации тестов.

Другой крайностью является игнорирование внешних компонентов при анализе. Вызовы внешних функций в таком сценарии заменяются грубыми аппроксимациями сверху или снизу. При использовании такого подхода за сокращение времени анализа приходится платить низкой полнотой или точностью обнаружения ошибок, или невысоким покрытием и низким качеством тестов при решении задачи генерации тестов.

Альтернативой двум крайним подходам является группа подходов, основанная на моделировании поведения внешних компонентов. Внешние компоненты (функции, библиотеки и т. п.) в таком случае заменяются в процессе анализа на их упрощенные модели.

Характеристики подходов в задачах обнаружения ошибок и генерации тестов в многокомпонентных программах приведены в табл. 1.

Рассмотрим основные используемые подходы в отдельности более детально.

Игнорирование внешних зависимостей

В инструменте статического анализа программ oo7 [1] в целях определения наличия угроз информационной безопасности и автоматической корректировки их поведения механизм анализа основывается на наборе правил упрощения и аппроксимации ряда аспектов исполнения анализируемой программы. Одним из таких правил является маркировка возвращаемого значения как небезопасного (подконтрольного злоумышленнику) для функций, анализ которых не может быть осуществлен, включая внешние библиотеки и другое стороннее ПО. Стандартная библиотека при этом выступает как подходящая для анализа зависимость, однако взаимодействие со средой исполнения остается за гранью анализа. Подобный подход, как это подчеркивается авторами инструмента [1],

■ ***Таблица 1.*** Подходы к анализу в задачах поиска ошибок в программах и генерации тестов
■ ***Table 1.*** Approaches to analysis in the tasks of finding errors and test generation

Подход	Метод повышения эффективности	Поиск ошибок			Генерация тестов		
		Время анализа	Полнота анализа	Точность анализа	Время генерации	Покрытие	Качество тестов
Анализ всего проекта с внешними компонентами	Ограничение глубины анализа	Очень высокое	Высокая	Средняя	Очень высокое	Высокое	Высокое
		Среднее / высокое	Средняя	Средняя	Среднее	Среднее	Среднее
	Ограничение времени анализа	Управляемое	Средняя	Средняя	Управляемое	Среднее	Среднее
Анализ проекта с игнорированием компонентов	Аппроксимация сверху	Низкое	Высокая	Очень низкая	Низкое	Низкое	Низкое
	Аппроксимация снизу	Низкое	Очень низкая	Высокая	Низкое	Низкое	Низкое
Анализ проекта с моделированием поведения компонентов	Управляемая аппроксимация поведения внешних компонентов	Среднее / низкое	Высокая	Управляемая	Среднее / низкое	Высокое	Высокое

приводит к появлению большого числа ложноположительных срабатываний, что негативно сказывается на практической полезности подобных инструментов и требует ручной корректировки ограничений и механизмов фильтрации при выполнении анализа.

CryptoGuard [2] — еще один пример инструмента, в котором игнорируются внешние зависимости. Вместо анализа зависимостей авторы с помощью набора алгоритмов уточнения пытаются выяснить, может ли используемый в конкретном месте вызов внешней библиотеки быть причиной уязвимости. Такой подход позволяет снизить количество ложноположительных срабатываний, однако, как отмечают сами авторы, это может приводить к ложноотрицательным результатам.

Полный анализ программы

Анализаторы в общей массе направлены на работу с подконтрольным пользователю кодом. При необходимости анализа зависимостей проекта есть возможность подключить их в качестве модулей или явно передать анализатору путь к ним. В таком случае код проекта и его зависимостей будет рассматриваться инструментом анализа как единое целое. Однако, как уже было отмечено, при полном анализе многокомпонентных проектов инструменты сталкиваются с взрывом числа состояний, который приводит к катастрофическому увеличению времени анализа. Следует также отметить, что не всегда подключаемые модули имеют представление, пригодное к анализу выбранным инструментом или методом, что также затрудняет поиск ошибок на практике.

К такого рода подходу при всей сложности исполнения можно прибегнуть лишь в случаях, когда внешние зависимости могут быть явно подключены и задействованы анализируемым ПО в рамках составленной модели. Данное ограничение исключает применение в качестве подобной «внешней» зависимости широко распространенные операционные системы общего назначения с богатыми возможностями для абстрагирования от аппаратного уровня, использующие специализированные статические или динамические программные модули для работы напрямую с аппаратным обеспечением среды исполнения. Функционирование же последнего, в свою очередь, может быть успешно смоделировано.

Встроенные механизмы анализа стандартных библиотек

Подавляющее большинство разработанных программных продуктов опирается в своей работе на некоторые модули и коллекции функций, считающиеся для целевой платформы и среды исполнения стандартными. Из этого прямо следует, что именно эти фрагменты программного кода будут использоваться наиболее часто и, как следствие, будут наиболее востребованы в процессе анализа. Необходимо также подчеркнуть, что упомянутые элементы нередко имеют нетривиальное поведение, которое требует создания упрощенной модели для эффективного использования. Для решения этой проблемы распространенным подходом можно назвать создание примитивных «заглушек» — фрагментов кода анализатора, отражающих очень упрощенное, грубое поведение функции или модуля [3].

Например, это могут быть жестко заданные последовательные проверки свойств ключевых параметров функции и возвращаемого значения. Подобным образом организована работа в компиляторе Clang (https://clang-analyzer.llvm.org) для языков C, C++ и Objective-C. Clang Static Analyzer по умолчанию содержит набор различных проверок. Это могут быть как проверки небезопасного использования API, проверки на основе каких-либо известных стандартов безопасного кодирования, например CERT (https://wiki.sei.cmu.edu/confluence/display/c/2+Rules), так и отдельно созданные пользователем модули. Анализ выполняется для пользовательского кода, а не для сторонних библиотек. Тем не менее если используемые зависимости были собраны при помощи Clang, то стандартные проверки были пройдены. Подобный подход обладает низкой гибкостью и эффективностью вследствие необходимости внесения изменений в инструмент анализа при изменении состава и содержания функций стандартной библиотеки. Кроме того, прямой перенос или адаптация «заглушек» невозможны по причине сильной привязки к методологии анализа, на которой базируется инструмент, и особенностей его программной архитектуры.

DSL-описания примитивных моделей

Некоторые средства анализа предоставляют механизмы для описания простых свойств внешних функций. Обычно это простые DSL, основанные на структурированных текстовых форматах. Например, инструмент анализа безопасности FlowDroid [4] поддерживает интерфейс для моделирования внешних библиотек в виде специализированных XML-файлов. Подобного рода правила (https://github.com/secure-software-engineering/FlowDroid/tree/develop/soot-infoflow-summaries/testSummaries) содержат примитивные описания правил taint-анализа (источники, подверженные стороннему влиянию, и потенциальные цели для проведения атак) и позволяют сократить и упростить процесс анализа потока данных. Инструмент изначально включает в себя некоторые предопределенные правила, которые обрабатывают классы коллекций, строковые буферы и аналогичные часто используемые структуры данных, указывая таким образом, например, что добавление испорченного элемента в набор искажает весь набор целиком. Если вызов метода библиотеки не имеет связанного правила, он анализируется полностью при наличии соответствующего ему байт-кода. Похожий подход реализуется в инструменте статического анализа Borealis [5]. В нем используются JSON-описания для задания паттернов доступа и работы с памятью внешних зависимостей на основе упрощенного подмножества языка ACSL [6], а также специализированные аннотации в секции комментариев в исходном коде анализируемых проектов (программ и (или) библиотек). Следует отметить, что данный подход к организации моделей позволяет проводить исследование проектов, программные модули которых не имеют представления, подходящего для целей анализа. В работе [7] описывается язык CrySL, созданный для составления простых спецификаций использования Java объектов. Основной акцент сделан на применении конечных автоматов для описания сценариев корректного использования криптографических алгоритмов и библиотек. Подобные описания полагается использовать в сочетании с представленным в той же работе $\text{CogniCrypt}_{\text{sast}}$ — инструментом статического анализа — для выявления ошибок в Android-приложениях. Применительно к описанию программных интерфейсов в индустрии широкое распространение получил стандарт OpenAPI Specification (https://spec.openapis.org/oas/latest.html, ранее — Swagger). В сущности, OpenAPI — это стандарт документации для описания API, нацеленный на обеспечение взаимопонимания между разработчиками различных систем и технологий. Этот стандарт предоставляет подробную информацию о том, как использовать API, включая определение структуры запросов, ответов, параметров, методов обработки ошибок, аутентификации и авторизации. Необходимо отметить, что данный стандарт позволяет в ограниченном виде применять конечно-автоматные модели (https://schemathesis.readthedocs.io/en/stable/stateful.html) для организации сервисов со скрытым внутренним состоянием. OpenAPI-спецификации формируются в виде YAML- или JSON-описаний.

Комплексные описания функций библиотек

В инструменте UnitTestBot Java [8], целью которого является генерация модульных тестов, применяются аппроксимации некоторых классов стандартной библиотеки Java, так как часть из них сложна для статического анализа и их обработка ведет к большому числу ложных срабатываний. Для преодоления этого в инструменте используется механизм «фиктивных» (mock) объектов, моделирующий поведения некоторых символьных примитивов. Аппроксимации написаны на языке Java и являются упрощенной версией классов стандартной библиотеки. Благодаря тому, что они являются частью инструмента анализа и написаны с учетом специфики применяемого механизма анализа, обеспечивается более быстрая и точная работа инструмента. В работе [9] описыва-

ется подход Dione с использованием ограниченного подмножества языка Python для создания спецификаций на базе конечных автоматов ввода-вывода. Подход предполагает использование специализированных аннотаций (декораторов) в сочетании с созданием классов с особой структурой (https://github.com/cyphyhouse/Dione/tree/master/system_tests/ioa_examples) для описания основных элементов автомата на языке Dione с последующей трансляцией в представление, пригодное для использования статическим анализатором Dafny (https://www.microsoft.com/en-us/research/publication/dafny-automatic-program-verifier-functional-correctness/).

Однако возможность применения аппроксимаций и спецификаций на подобной основе ограничена выбранным языком написания. Кроме того, при написании подобных оболочек необходимо искусственно контролировать использование различных конструкций, так как язык общего назначения, такой как, например, Java или Python, обладает гораздо большей выразительностью, чем того требует аппроксимация. Ошибка разработчика при создании такой упрощенной модели (например, использование циклов с неизвестным числом итераций или явных или неявных рекурсий) может привести к взрыву пространства состояний при анализе и, как следствие, к критическому увеличению времени его проведения.

Моделирование библиотек на основе спецификаций

Подходы, использующие спецификации библиотек и библиотечных функций, активно развиваются и находят свое применение в различных средствах анализа программ. По способу организации спецификаций их можно разделить на две подгруппы: подходы на основе встроенных спецификаций и подходы на основе внешних спецификаций.

Подходы на основе *встроенных спецификаций* предполагают, что спецификации библиотек и библиотечных функций встраиваются внутрь исходного текста самих программных библиотечных модулей. Для этого применяются механизмы структурированных комментариев и аннотаций в зависимости от особенностей применяемого средства моделирования и (или) представления. Одним из примеров такого описания спецификации является язык JML [10]. Он позволяет описывать поведения в виде логических выражений (высказываний, пред- и постусловий, инвариантов) для отдельных методов и полей на основе специальным образом оформленных комментариев в исходном тексте модуля на языке Java. Концептуально идентичными средствами моделирования обладают также инструменты, базирующиеся на языках, схожих с ACSL, такие как Mopsa [11] или C-kernel [12], отличительной особенностью которых является фокус на языках семейства C и на работе с указателями и моделями памяти.

К недостаткам этих инструментов можно отнести невозможность поведенческого описания в явном виде, а также привязанность к окружению определенного языка программирования. Для таких спецификаций имеется одно существенное ограничение, связанное с необходимостью доступа к исходному коду. Как следствие, не всегда возможно встроить спецификацию в код библиотеки, а аппроксимация больших библиотек затруднительна в силу необходимости, зачастую, повторять уже реализованное поведение, тем самым значительно увеличивая объем кодовой базы и повышая вероятность возникновения ошибок. Но даже при наличии исходного кода любое обновление внешней библиотеки приведет к необходимости переноса и переработки спецификации.

Подходы на основе *внешних спецификаций* предполагают создание спецификации библиотеки или отдельных ее функций в виде отдельных артефактов. Такой подход реализован в языке SLIC [13]. SLIC предназначен для описания программ на языке C, он позволяет в близких к C синтаксисе и семантике описывать спецификации без модификаций исходного кода. Поддерживается машина состояний, пред- и постусловия. Спецификация компилируется в C, позволяет выполнять инструментирование и отслеживать работу программы в виде трасс. Схожим образом построено взаимодействие анализатора с языком MetaL [14]. Спецификация MetaL исполняется с помощью компилятора XGCC и позволяет проверять выполнение определенных свойств, ограничений и соответствие переходам в машине состояний. Несмотря на потенциальную возможность абстрагироваться от целевого языка программирования, SLIC и MetaL ориентированы на совместное использование с языком C, и для их применения используются специальные препроцессоры.

К подходу на основе внешних спецификаций относится также язык CASL [15]. Он позволяет задать представленную в логике первого порядка алгебраическую спецификацию, предназначенную для описания поведения и структуры программ. В основе подхода лежит функциональная парадигма программирования. Для CASL реализован набор инструментов, позволяющий взаимодействовать со спецификацией для выполнения необходимых задач, однако функциональная природа полученных моделей значительно отличается от парадигм и методологий,

применяемых современными промышленными языками общего назначения.

Очевидными достоинствами подходов на основе внешних спецификаций являются:

— отсутствие необходимости доступа к исходным кодам библиотеки;

— отсутствие необходимости внесения изменений в чужой программный код;

— возможность абстрагирования спецификаций от целевого языка программирования;

— возможность управления уровнем детализации моделирования.

Выводы

Перечисленные подходы в разной степени решают проблему моделирования сторонних библиотек. При этом реализации, использующие тот или иной подход, имеют специфичные для него недостатки: большое время анализа, низкую точность, недостаточную универсальность. Некоторые системы, такие как, например, SharpChecker [16], одновременно сочетают несколько подходов (встроенные определения, коллекцию примитивных описаний, аппроксимацию на целевом языке) с целью повысить полноту и точность анализа, но при этом также не позволяют создавать гибкие аппроксимации разного уровня детализации. Цель данной работы — предложить универсальный, не зависящий от языка программирования подход, позволяющий моделировать библиотеки с разной степенью точности в зависимости от задач анализа.

Предлагаемый подход к моделированию библиотек

Отсутствие каких-либо сведений об используемой библиотеке сильно снижает точность анализа, увеличивая количество обнаруженных ложноположительных или ложноотрицательных дефектов. В то же время анализ всех зависимостей вместе с программой проверки приводит к взрыву состояний и критическому росту времени анализа, вследствие чего его применение становится нецелесообразно.

Одним из возможных путей решения указанных проблем является моделирование поведения библиотеки с использованием формальных спецификаций. Основная идея подхода состоит в создании механизма, позволяющего разработчику описывать видимое извне поведение библиотечных функций, используя ограниченные выразительные средства. Основные требования, предъявляемые к такому механизму:

— выразительность, достаточная для описания возможностей большинства существующих библиотек;

— поддержка основных возможностей процедурных и объектно-ориентированных языков программирования;

— ограниченная вычислительная сложность, не позволяющая создавать спецификации, способные привести к взрыву числа состояний системы при анализе;

— возможность раздельного хранения спецификации и библиотеки.

В рамках данной работы для написания спецификаций использовался язык LibSL [17]. Этот язык, уже много лет развиваемый авторами статьи, показал свою эффективность в решении нескольких задач программной инженерии, связанных с описанием свойств и моделированием поведения программных библиотек. При помощи его можно создавать описания поведения различных функций и методов в семантике, близкой к целевому языку программирования. Предлагаемый в данной работе подход к моделированию поведения объектов программной библиотеки основывается на ручном создании аппроксимированного, но при этом наиболее приближенного к оригиналу внутреннего модельного поведения публичных элементов библиотеки (соответствующих семантических отображений типов данных, функций и методов). Ключевым элементом является использование наборов специализированных семантических действий (ключевое слово *action* в языке LibSL), свойственных целевой среде исполнения модели. Дополнительные семантические действия позволяют выполнять моделирование таких элементов и взаимодействий, как циклы (обычные и индексированные), захват и выброс исключений, вызов методов с динамической диспетчеризацией (интерфейсы и абстрактные типы данных), обращение к примитивам синхронизации и пр.

Необходимо отметить, что в языке LibSL применяется комбинированный подход к организации функциональных блоков: публичные функции, обеспечивающие доступ к функциональности библиотеки внешним приложениям для прямого обращения и применяемые на этапе связывания с основной программой и другими библиотеками, группируются в рамках какого-либо конечного автомата (ключевое слово *automaton*), выполняющего моделирование состояния экземпляра сущности какого-то одного выбранного семантического типа данных. Примером такой сущности может быть как структура и статические функции, описывающие работу с файлом [17], так и динамический список элементов или публичный интерфейс, предоставляющий доступ к базе данных.

Однако напрямую использовать созданные таким образом спецификации затруднительно, поэтому необходима трансляция на промежуточ-

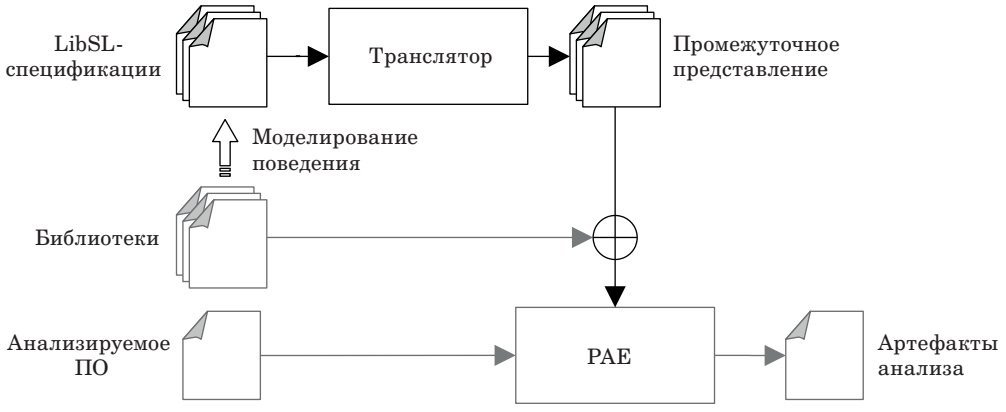
ный исполняемый язык программирования, для которого уже существуют инструменты анализа программ. Для решения этой проблемы был создан инструмент, основными задачами которого являются: разбор входных спецификаций; поиск примитивных семантических ошибок; преобразование в файловые объекты в представлении, пригодном для использования целевым инструментом анализа (program analysis engine, PAE). В рамках данной работы основным выходным представлением являлись исходные тексты объектных классов на языке Java, взаимодействующих с PAE посредством специализированного программного интерфейса (API). В дополнение к этому был разработан, как часть вспомогательного инструментария, комплект вспомогательных методов, реализующих работу с примитивными типами данных, и другие функции общего назначения (сериализация, поиск элемента коллекции, проверка эквивалентности коллекций и их элементов и т. д.). Общая архитектура предлагаемого подхода представлена на рис. 1.

Поведение вспомогательных программных модулей (библиотек) частично или полностью описывается в форме коллекции LibSL-спецификаций. Далее, при помощи утилиты-транслятора, выполняется преобразование их в требуемое (целевое) представление, пригодное к использованию выбранным инструментом PAE. Представленная архитектура опирается на существование в PAE механизма, позволяющего подменять (статически или динамически) внешние зависимости анализируемого ПО на предоставляемые модели библиотек при их наличии. В случае же отсутствия аппроксимаций предполагается использовать оригинальные программные модули или выполнить их автоматическое упрощение при наличии возможности.

Для демонстрации принципов организации предлагаемого подхода и механизмов работы инструмента рассмотрим пример использования циклической конструкции и ее реализации в сочетании с использованием PAE, имеющего встроенную поддержку этой конструкции.

Модель поведения для метода *lastIndexOf* класса *ArrayList* из пакета *java.util* стандартной библиотеки языка Java изображена на рис. 2, а соответствующая ей реализация на целевом языке программирования — на рис. 3 (числовые метки соответствуют элементам рис. 2). Оригинальный класс *ArrayList* предназначен для выполнения операций, свойственных динамическому списку элементов, индексированному целыми неотрицательными числами. Назначение метода *lastIndexOf* заключается в поиске порядкового номера элемента (отсчет ведется с нуля), эквивалентного объекту, который был передан в качестве аргумента. В случае неудачи (список пуст или эквивалентных элементов найдено не было) метод возвращает значение –1. Модель такого поведения описывается на языке LibSL (см. рис. 2) функцией (ключевое слово *fun*) с названием, соответствующим названию исходного метода. Сигнатура функции аналогична сигнатуре оригинального метода, однако дополнительно включает указание на экземпляр объекта класса *ArrayList* (параметр *self*). Для указания объекта, для типа которого предназначено описание моделирующего конечного автомата, используется аннотация *@target*, позволяющая применять сокращенный синтаксис при обращении к автоматным переменным, таким как, например, *storage*.

Меткой *1* (см. рис. 2) выделены встроенные типы данных специального назначения и сопутствующие им наборы примитивных действий, позволяющие повысить эффективность процесса анализа модели, используя механизмы и модели, поставляемые в конкретной реализации целевого PAE. В рамках данной работы такими



■ ***Рис. 1.*** Общая схема архитектуры подхода
■ ***Fig. 1.*** General architecture outline

```
18   automaton ArrayListAutomaton
19   (
20       var storage: list<Object>
21   )
22   : ArrayList
23   {
     ...
761  fun *.lastIndexOf (@target self: ArrayList, o: Object): int
762  {
763      result = -1;                                    (1)
764      val size: int = action LIST_SIZE(this.storage);
765      if (size != 0)                                  (2)
766      {
767          action ASSUME(size > 0);
768          var i: int = 0;
769          action LOOP_FOR(                            (3)
770              i, size - 1, -1, -1,
771              lastIndexOf_loop(i, o, result)
772          );
773      }
774  }
775
776  @Phantom proc lastIndexOf_loop (i: int, o: Object, result: int): void
777  {
778      val e: Object = action LIST_GET(this.storage, i);
779      if (action OBJECT_EQUALS(o, e))                 (4)
780      {
781          result = i;
782          action LOOP_BREAK();
783      }                                               (5)
784  }

     ...
1002 }
```

■ ***Рис. 2.*** Описание циклической конструкции средствами языка LibSL
■ ***Fig. 2.*** Description of a loop in LibSL

```
743  /**
744   * [FUNCTION] ArrayListAutomaton::lastIndexOf(ArrayList, Object) -> int
745   * Source: java/util/ArrayList.main.lsl:761
746   */
747  public int lastIndexOf(Object o) {
748      int result = 0;
749      Engine.assume(this.__$lsl_state == __$lsl_States.Initialized);
750      /* body */ {
751          result = -1;                                (1)
752          final int size = this.storage.size();
753          if (size != 0) {
754  (2)         Engine.assume(size > 0);
755              int i = 0;
756              for (i = size - 1; i > -1; i += -1) {
757  (3)             final Object e = this.storage.get(i);
758                  if (LibSLRuntime.equals(o, e)) {
759                      result = i;                     (4)
760                      break;
761                  }                                   (5)
762              }
763              ;
764          }
765      }
766      return result;
767  }
```

■ ***Рис. 3.*** Реализация циклической конструкции. Результат трансляции на язык Java
■ ***Fig. 3.*** Implementation of a loop. Translation result (Java)

типами данных являются «список» (*list*<*E*>, где *E* — тип элемента списка; *list*<*Object*> на рис. 2) и «сопоставление» (*map*<*K*, *V*>, где *K* и *V* — некоторые типы данных, являющиеся ключом и значением сопоставления соответственно). Поддерживаемые операции ограничиваются возможностями предоставляемого интерфейса взаимодействия со средой исполнения PAE и включают такие действия, как: создание пустой коллекции, добавление элемента, проверка вхождения ключа в коллекцию, определение количества элементов в коллекции, получение элемента (по ключу или индексу), удаление сопоставления «ключ=значение». Дополнительно

была организована поддержка таких операций, как пересечение и объединение ключей для пары сопоставлений с выборкой значений по одному из сопоставлений, а также получение случайного существующего ключа. Стоит отметить, что в рамках реализации трансляции на язык Java (см. рис. 3) для одновременной поддержки моделирования поведения классов *java.util.IdentityHashMap* и *java.util.HashMap* (и им подобных) была реализована прослойка, состоящая из интерфейса контейнера и двух частных реализаций: с обращением к методам *hashCode* и *equals* (для *HashMap* и *LinkedHashMap*) и без с использованием операции сравнения ссылок на объекты ключей (для *IdentityHashMap*).

Меткой *2* на рис. 2 отмечены действия, являющиеся специфичными для подхода к моделированию исполнения, применяемого целевым PAE, позволяющие повысить эффективность процесса анализа, сократив исследуемое пространство состояний. Выражение *ASSUME*, в частности, широко применяется при моделировании элементов стандартной библиотеки, частично или полностью полагающихся на способность PAE оперировать символьными переменными. В примере на рис. 2 использование этого действия позволяет уменьшить время анализа для случаев, когда PAE примет допустимым отрицательное значение для переменной *size*.

Метками *3* и *5* отмечены действия, изменяющие порядок исполнения операций. В примере на рис. 2 действие *LOOP_FOR* используется для моделирования цикла со счетчиком для перебора элементов списка, вмещающего в себя отдельные элементы (поле *storage*), и для поиска элемента, эквивалентного заданному (входной параметр "*o*"). Параметрами действия являются: переменная-счетчик, начальное (включительно) и конечное (исключительно) значение счетчика, приращение счетчика на каждой итерации. На рис. 3 показано, что условие остановки цикла организовано с использованием оператора «больше». Выбор оператора основывается на значении приращения для счетчика: для константного числового значения результат выбора зависит от знака, иначе, если константное значение не удалось определить, используется оператор «не равно». Для моделирования тела цикла используется вызов подпрограммы *lastIndexOf_loop* (ключевое слово *proc*). При выполнении трансляции действия *LOOP_FOR* вместо тела цикла подставляются выражения из вызываемой подпрограммы. Переменные, которые требуется использовать внутри тела цикла, описываются как параметры используемой подпрограммы (переменные "*i*", "*o*" и "*result*" на рис. 2 и 3).

Меткой *4* отмечены действия общего назначения, являющиеся широко применимыми, такие, как, например, сравнение двух значений на эквивалентность. В рассматриваемом примере образцом такового является *OBJECT_EQUALS*. Это действие преобразуется в вызов внешнего статического метода и выполняет сравнение, опираясь на тип передаваемых параметров в соответствии с правилами целевого языка (Java в данном случае). Приведенная реализация позволяет проводить модификацию поведения указанного действия независимо от преобразованного набора спецификаций, однако ее недостатком является использование вызова статического метода, что может оказаться затратным для некоторых видов анализа.

Аннотация *@Phantom* позволяет предотвратить появление переменной, функции или подпрограммы в результирующем выводе. Подобного рода функциональность дает возможность, в сочетании с механизмом подстановки выражений, как уже было упомянуто, описывать и использовать исполнимое тело динамической структуры, такой как цикл, лямбда-выражение или иной функтор. Очевидным достоинством такого подхода является контроль захвата и использования переменных и других объектов внутри исполняемого фрагмента. Вторым преимуществом является возможность переиспользования одной реализации в разных участках одной спецификации. Недостатком же, в свою очередь, является повышение разрозненности кодовой базы спецификаций вследствие разделения места использования от места объявления подпрограммы.

Результатом трансляции для вышеуказанных действий является вызов соответствующего метода для переменной, переданной в качестве одного из параметров действия. Исключения составляют действия, назначения которых схожи с метками *3* и *5* рис. 2, порождая синтаксическую конструкцию, соответствующую специфике и возможностям PAE и целевого представления (метки *3* и *5* на рис. 3).

Как уже сказано, одним из отличительных преимуществ использования языка LibSL является поддержка описаний конечных автоматов. Реализовать поддержку конечных автоматов в промежуточном представлении возможно различными способами при наличии средств, позволяющих моделировать чтение текущего состояния, сверку состояния с фиксированным значением, сохранение нового значения состояния. Дополнительные действия зависят от возможностей выбранного промежуточного состояния, среды моделирования и целей анализа.

Показан вызов статического метода *Engine.assume* в начале метода аппроксимации (строка 749 на рис. 3). В данном случае это позволяет ограничить исследование возможных путей ис-

полнения только теми, при которых аппроксимация (автомат) была инициализирована корректно.

Моделирование исполнения осуществляется в рамках поддержки taint-анализа с применением виртуальных семантических меток (рис. 4). Для работы с метками организован набор соответствующих семантических действий, позволяющих добавить, проверить наличие (*HAS_MARK*), удалить метку, сигнализировать об обнаруженной ошибке (действие *SINK_ALARM*). С использованием этих действий выполняется контроль текущего состояния путем проверки наличия метки состояния, исполнение функции в котором не предусмотрено спецификацией. Данная функциональность требует от пользователя лишь выполнить сопоставление функций с состояниями конечного автомата через группы переходов (ключевое слово *shift*). Механизмы проверки и изменения состояния будут добавлены автоматически на этапе трансляции (рис. 5).

```
18  automaton fileIo_Stream(): fileIo_Stream
19  {
20      initstate Open;
21      finishstate Closed;
22
23      shift Open -> self   by [ readSync, ... ];
24      shift Open -> Closed by [ closeSync ];
    ...
67  fun *.readSync(@target self: fileIo_Stream,
68                     buffer: ArrayBuffer,
69                     @nullable options: ReadOptions
70                 ): number
71  {
72      if (action HAS_MARK(self, TM_FILE_WRITEONLY))
73          action SINK_ALARM(ERR_READ_FROM_WRITEONLY);
74
75      action COPY_MARKS_ALL(buffer, result);
76  }
    ...
```

■ ***Рис. 4.*** Описание метода и автоматной модели
■ ***Fig. 4.*** Description of a method and associated automaton model

```
    ...
67  fun *.readSync(@target self: fileIo_Stream,
68                     buffer: ArrayBuffer,
69                     @nullable options: ReadOptions
70                 ): number
71  {
        if (action HAS_MARK(self, `TM-fileIo_Stream-#-Closed`))
            action SINK_ALARM(`<err:state>fileIo_Stream.readSync`);

72      if (action HAS_MARK(self, TM_FILE_WRITEONLY))
73          action SINK_ALARM(ERR_READ_FROM_WRITEONLY);
74
75      action COPY_MARKS_ALL(buffer, result);
76  }
    ...
```

■ ***Рис. 5.*** Коррекция описания метода по автоматной модели
■ ***Fig. 5.*** Automatic correction of the method description

Использование конечного автомата, описанного выше, с помощью синтаксиса создания нового экземпляра автомата-объекта продемонстрировано на рис. 6.

Видно, что конструкция, обозначающая создание нового экземпляра автомата и выражение присваивания значения переменной *result*, заменяется вызовом действия, добавляющего метку состояния автомата к указанной переменной. Необходимо отметить, что в приведенном примере используются метки без применения параметризации, что значительно сужает спектр возможных автоматных состояний из-за ограничений только значениями псевдопеременной *state*, заданными при объявлении структуры автомата (ключевые слова *state*, *initstate*, *finishstate*). При использовании более функционального средства моделирования потенциально возможно описывать более сложные состояния, применяя дополнительные пользовательские автоматные переменные.

Экспериментальные исследования предлагаемого подхода

Для оценки применимости предлагаемого подхода и корректности его реализации было проведено два эксперимента. Цель первого экс-

```
17  automaton fileIo(): fileIo
18  {
    ...
31  static fun *.createStreamSync (path: string, mode: string): fileIo_Stream
32  {
33      if (mode == "r")                action ADD_MARK(result, TM_FILE_READONLY);
34      if (mode == "w" || "a" == mode) action ADD_MARK(result, TM_FILE_WRITEONLY);
35
36      if (action VALUE_CONTAINS(path, "../"))
37          action SINK_ALARM(CWE_23);
38
39      action COPY_MARKS_ALL(path, result);
40
41      result = new fileIo_Stream(state = Open); // -> action ADD_MARK(result, `TM-fileIo_Stream-#-Open`);
42  }
    ...
```

■ ***Рис. 6.*** Применение автомата
■ ***Fig. 6.*** Automaton usage example

перимента — проверка корректности создания спецификаций классов коллекций стандартной библиотеки Java. Второй эксперимент проводился в целях анализа эффективности предложенного подхода моделирования стандартной библиотеки.

Корректность поведенческих аппроксимаций

Первый эксперимент был поставлен с целью убедиться, что сгенерированные по LibSL-спецификациям классы Java имеют поведение, максимально приближенное к поведению оригинальных классов. Множество классов для моделирования было ограничено только некоторыми из представленных для работы с коллекциями из пакета *java.util*, так как стандартная библиотека языка Java предлагает программисту очень широкий спектр возможностей для реализации различных программных систем, и полное ее исследование существенно превысило бы рамки журнальной статьи. Тем не менее наиболее сложными для моделирования и наиболее часто используемыми классами стандартной библиотеки Java являются коллекции, что подтверждается данными из статьи [18]. Поэтому для проведения эксперимента авторы ограничились только этой частью спецификаций.

Эксперимент заключался в следующем. С помощью генератора тестов Randoop (https://randoop.github.io/randoop/) создавался набор тест-кейсов для определенного класса коллекций Java. После этого создавалась новая версия программы, в которой при помощи технологии Java Agent обращения к оригинальным классам коллекций динамически заменялись на обращения к сгенерированным классам аппроксимаций. В итоге сравнению поведения подвергались две версии программ: с классами исходной библиотеки и со смоделированными классами. Все сгенерированные тесты успешно были скомпилированы и запущены в среде разработки Intellij Idea, что подтверждает их синтаксическую корректность. Ошибки другого типа в тестах не являются критичными, так как тесты используются исключительно для сравнения поведения эталонной и модельной реализаций.

Следует отметить следующие особенности эксперимента: в рамках разработанных спецификаций классы *java.util.HashSet* и *java.util.LinkedHashSet* имеют одинаковое поведение: они построены на структурах, которые, в отличие от оригинального класса *java.util.HashSet*, всегда выдают элементы в порядке их добавления. В рамках работ по моделированию стандартной библиотеки Java было принято решение применить такую аппроксимацию: пренебречь порядком элементов. С учетом данного допущения все сгенерированные тест-кейсы с помощью Randoop могут быть пройдены, если создавать их для всех методов класса *java.util.LinkedHashSet*, за исключением конструктора *HashSet(int, float, boolean)*, который позволяет изменить порядок доступа к элементам. В этом случае при вызове метода *toString* порядок всегда сохраняется, как в классе *java.util.LinkedHashSet*. Поэтому тесты генерировались для *java.util.LinkedHashSet* и *java.util.LinkedHashMap*, а заменялись на *generated.java.util.HashSet* и *generated.java.util.HashMap*.

На текущем этапе разработки инструментария парсер и транслятор не предоставляют средств для работы с объявлениями обобщенных типов данных. Их поддержка будет добавлена в будущем. Вследствие этого метод *toArray* класса *java.util.LinkedList* (и других обобщенных коллекций) после выполнения трансляции имеет немного отличающуюся сигнатуру. В оригинальном классе его объявление применяет типовой параметр (рис. 7, строка *A*), а в сгенерированном классе метод применяет тип *java.lang.Object* (рис. 7, строка *B*). На работу применяемого инструмента статического анализа данное отличие не оказало большого влияния, однако это различие необходимо учитывать при использовании других инструментов.

Следует отметить, что реализация аппроксимации данного метода намеренно нарушает правила типизации языка Java: результирующий массив состоит из ссылок на элементы типа *java.lang.Object* вместо обращения к методам *java.lang.Class.getComponentType()* и *java.lang.reflect.Array.newInstance()*. При использовании аппроксимаций по прямому назначению (с применением PAE) данное отличие не оказало существенного влияния на результаты моделирования и анализа. По этой причине этот метод был исключен из генерации тест-кейсов при использовании утилиты Randoop. Иначе это приводило к выбросу исключения *java.lang.ClassCastException*. Пример теста, демонстрирующего ошибку, показан на рис. 8.

Для проведения эксперимента использовалась среда исполнения OpenJDK и инструмент Randoop (версии 4.3.2). При генерации тест-кейсов с помощью Randoop использовался детерминированный подход. Параметры запуска генератора тестов и список целевых методов доступны публично (https://github.com/vpa-

```
A)  public <T> T[]  toArray(T[]      a) { ... }
B)  public Object[] toArray(Object[] a) { ... }
```

■ ***Рис. 7.*** Вариации сигнатуры метода класса LinkedList

■ ***Fig. 7.*** Method signature variations of the LinkedList class

```
1  LinkedList<String> list = new LinkedList<>();
2  String[] empty = new String[0];
3  String[] array = list.toArray(empty); // <- error
   ...
```

■ ***Рис. 8.*** Упрощенный фрагмент исходного кода генерируемого теста

■ ***Fig. 8.*** Simplified generated test source code

research/jsl-spec-validation). В табл. 2 приведены результаты эксперимента. Как видно из таблицы, количество удачно выполненных тест-кейсов почти для всех коллекций 100 %. Однако некоторые тест-кейсы для класса *java.util.Optional* завершились ошибкой. Это связано с вызовом метода *getClass*, который возвращает внутреннее имя класса JVM. В указанных тест-кейсах он вместо ожидаемого оригинального класса *java.util.Optional* возвращает имя подмененного с помощью java agent класса, т. е. *generated.java.util.Optional*. В рамках данного эксперимента это отличие не является критическим, так как не влияет на функциональность сгенерированного кода при использовании в составе PAE. Сценарий непройденных тест-кейсов выглядит следующим образом (рис. 9).

■ ***Таблица 2.*** Оценка корректности моделей

■ ***Table 2.*** Evaluation of the correctness of the models

Имя класса	Общее количество сгенерированных тест-кейсов	Удачно выполненные тест-кейсы	Неудачно выполненные тест-кейсы
java.util.ArrayList	1099	1099	0
java.util.LinkedList	1065	1065	0
java.util.LinkedHashSet	1042	1042	0
java.util.LinkedHashMap	987	987	0
java.util.Optional	1017	1013	4
java.util.OptionalInt	1100	1100	0
java.util.OptionalLong	1106	1106	0
java.util.OptionalDouble	1117	1117	0

```
1  Optional<?> optA = Optional.of(1);
2  Class<?>   clazz = optA.getClass();
3  Optional<?> optB = Optional.of(clazz);
4  String       str = optB.toString();
5  assertEquals(
6      "'" + str + "' != 'Optional[class java.util.Optional]'",
7      str,
8      "Optional[class java.util.Optional]"
9  );
```

■ ***Рис. 9.*** Упрощенный исходный код генерируемого теста

■ ***Fig. 9.*** Simplified source code of a generated test

Из приведенного эксперимента видно, что поведение классов коллекций, сгенерированных по разработанным спецификациям, идентично поведению оригинальных классов из JDK.

Эффективность подхода

Второй эксперимент был поставлен с целью оценить эффект от использования аппроксимаций поведения при выполнении анализа программ. Эксперимент заключался в следующем: был произведен запуск символьной виртуальной машины USVM (https://github.com/UnitTestBot/usvm) в режиме генерации тестов для классов пакета *org.apache.commons.collections4.set* проекта Apache Commons-collections (https://github.com/apache/commons-collections/tree/commons-collections-4.5.0-M1-RC1). Запуск производился в двух конфигурациях: с применением аппроксимаций, полученных после трансляции LibSL-спецификаций, и без них.

Результаты эксперимента представлены в табл. 3. Для каждого класса были получены такие характеристики, как количество методов, среднее покрытие по методам и общее время анализа. Из рассмотрения исключены конструкторы для данных классов, являющиеся тривиальными или явно (и только) использующие методы, анализ которых уже проводился отдельно. Показатель покрытия оценивался USVM автоматически на основе выполненных инструкций. Также было вычислено сокращение покрытия и сокращение времени анализа.

Полученные данные демонстрируют сокращение общего времени анализа более чем на 30 %, сокращение времени анализа для каждого класса не менее чем на 10 %, а также поддержание приемлемого (3 %) сокращения общего уровня покрытия при использовании LibSL-аппроксимаций. Значительное падение показателя покрытия для некоторых методов было вызвано применением неточных аппроксимаций в контексте малого количества методов в классе, что может быть ликвидировано повышением точности модели. При этом необходимо отметить, что при проведении экспериментов с усилением временных ограничений на анализ каждого метода до 30 с значение показателя «Покрытие» сохраняется, а сокращение времени анализа попрежнему остается на уровне 30 %.

Разработанные один раз аппроксимации функций стандартной библиотеки могут в даль-

■ ***Таблица 3.*** Оценка эффективности применения аппроксимаций
■ ***Table 3.*** Evaluation of the effectiveness of using approximations

Имя класса (число методов в классе)	Без аппроксимаций		С применением аппроксимаций		Сокращение	
	Покрытие, %	Время анализа, с	Покрытие, %	Время анализа, с	покрытия, %	времени анализа, %
CompositeSet (23)	84,04	634,3	78,74	327,8	5,30	48,32
ListOrderedSet (19)	91,05	364,3	86,58	205,6	4,47	43,56
MapBackedSet (18)	99,22	424,0	90,00	289,1	9,22	31,82
PredicatedNavigableSet (12)	100,00	349,9	100,00	244,1	0,00	30,24
PredicatedSet (3)	100,00	75,4	100,00	1,8	0,00	97,61
PredicatedSortedSet (7)	100,00	265,5	100,00	218,5	0,00	17,70
TransformedNavigableSet (13)	99,15	349,9	97,54	273,0	1,61	21,98
TransformedSet (4)	100,00	92,2	92,00	0,5	8,00	99,46
TransformedSortedSet (8)	98,63	287,2	96,00	219,9	2,63	23,43
UnmodifiableNavigableSet (19)	100,00	366,9	100,00	325,0	0,00	11,42
UnmodifiableSet (9)	100,00	28,5	94,44	0,1	5,56	99,65
UnmodifiableSortedSet (12)	100,00	152,5	100,00	131,3	0,00	13,90
Общее	**97,67**	**3390,6**	**94,61**	**2236,7**	**3,07**	**34,03**

нейшем неоднократно применяться при анализе проектов, использующих стандартную библиотеку, а таких проектов — большинство. Таким образом, временные затраты на создание моделей классов и функций многократно окупятся. Кроме того, очевидный выигрыш использования моделей проявляется также в ситуациях, когда анализ исходных проектов не может завершиться либо из-за чрезмерной сложности этих проектов, либо из-за слишком жестких ограничений по времени, накладываемых на анализ.

Ограничения подхода и дальнейшее развитие

Как было отмечено выше, спецификации, представленные отдельно от кода моделируемой функциональности, требуют правки при появлении изменений в оригинальном поведении и интерфейсе, а также при новых возможностях и оптимизациях, предоставляемых применяемым целевым представлением и инструментами анализа.

Кроме того, одним из ограничений к широкому применению представленного подхода к моделированию поведения библиотек является необходимость ручного создания спецификаций, что считается рутинной работой и требует особого внимания при описании моделей. Для автоматизации этого процесса уже реализованы инструменты (https://github.com/vpa-research/jsl-spec-skeleton-generator), позволяющие генерировать заготовки LibSL-спецификаций на основе интерфейсных объявлений классов и их публичных элементов. Частичной автоматизации можно добиться, применив инструменты, базирующиеся на восстановлении упрощенной модели [19], с последующим формированием человеко-читаемой формы, используя, например, технологии машинного обучения, как это было сделано для языков ACSL [20] и JML [21], на основе подстройки существующей искусственной нейронной сети [22].

Стоит отметить, что сам язык LibSL продолжает активно развиваться. В нем появляются дополнительные возможности для описания все более сложных структур данных и различных взаимодействий для обобщенных и специализированных типов данных. Развиваются также и инструменты языковой поддержки в современных средах разработки (https://plugins.jetbrains.com/plugin/23222-libsl-support).

Заключение

В статье описаны основные подходы к моделированию внешних библиотек, а также предлагается новый подход на основе спецификаций на базе языка LibSL. Этот подход позволяет моделировать поведение библиотек с разной степенью точности в зависимости от задач анализа. Также представлены экспериментальные исследования, показывающие эффективность предла-

гаемого подхода. Полученные модели продемонстрировали сокращение времени анализа для каждого метода при сохранении приемлемого уровня покрытия автоматически сгенерированных тестов. Однако есть и некоторые ограничения: необходимость выполнять трансляцию спецификаций в некоторое промежуточное представление, требование их актуализации (моделей и артефактов трансляции) при появлении изменений в поведении и интерфейсе, а также ограниченность возможностей для автоматизации процесса создания спецификаций. В целом предлагаемый подход представляет собой универсальное масштабируемое, не зависящее от языка программирования решение по моделированию внешних библиотек.

Литература

1. **Wang G., Chattopadhyay S., Gotovchits I., Mitra T., Roychoudhury A.** oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2019, vol. 47, no. 11, pp. 2504–2519. doi:10.1109/TSE.2019.2953709
2. **Rahaman S., Xiao Y., Afrose S., Shaon F., Tian K., Frantz M., Yao D., Kantarcioglu M.** Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. *Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security*, New York, NY, USA, 2019. Association for Computing Machiner, 2019, pp. 2455–2472. doi:10.1145/3319535.3345659
3. **Madsen M., Livshits B., Fanning M.** Practical static analysis of JavaScript applications in the presence of frameworks and libraries. *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2013. Association for Computing Machinery, 2013, pp. 499–509. doi:10.1145/2491411.2491417
4. **Arzt S., Rasthofer S., Fritz C., Bodden E., Bartel A., Klein J., Le Traon Y., Octeau D., McDaniel P.** Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 2014, vol. 49, no. 6, pp. 259–269. doi:10.1145/2666356.2594299
5. **Akhin M., Belyaev M., Itsykson V.** *Borealis Bounded Model Checker: The Coming of Age Story*. Present and Ulterior Software Engineering. M. Mazzara, B. Meyer (eds). Springer, Cham, 2017, pp. 119–137. doi:10.1007/978-3-319-67425-4_8
6. **Delahaye M., Kosmatov N., Signoles J.** Common specification language for static and dynamic analysis of C programs. *28th ACM Symp. on Applied Computing*, New York, NY, USA, 2013. Association for Computing Machinery, 2013, pp. 1230–1235. doi:10.1145/2480362.2480593
7. **Krüger S., Spath J., Ali K., Bodden E., Mezini M.** Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, 2019, vol. 47, no. 11, pp. 2382–2400. doi:10.1109/TSE.2019.2948910
8. **Ivanov D., Menshutin A., Fokin D., Kamenev Y.** UTBot java at the SBST2022 tool competition. *Proc. of the 15th Workshop on Search-Based Software Testing*, Pittsburgh, PA, USA, 2022. IEEE, 2022, pp. 39–40. doi:10.1145/3526072.3527529
9. **Hsieh C., Mitra S.** Dione: A protocol verification system built with Dafny for i/o automata. *Integrated Formal Methods: 15th Intern. Conf.*, Bergen, Norway, 2–6 December 2019. Springer International Publishing, 2019, pp. 227–245. doi:10.1007/978-3-030-34968-4_13
10. **Huisman M., Ahrendt W., Bruns D., Hentschel M.** *Formal Specification with JML*. Karlsruhe, Karlsruhe Institute of Technology, 2014. 51 p.
11. **Ouadjaout A., Miné A.** A library modeling language for the static analysis of C programs. *Intern. Static Analysis Symp.*, Springer, Cham, 2020, pp. 223–247. doi:10.1007/978-3-030-65474-0_11
12. **Промский А. В.** Верификация Си-программ: объяснение условий корректности и стандартная библиотека. *Моделирование и анализ информационных систем*, 2011, т. 18, № 4, с. 157–167. doi:10.3103/S0146411612070127
13. **Ball T., Rajamani S. K.** *SLIC: A Specification Language for Interface Checking (of C).* Technical Report MSR-TR-2001-21, Microsoft Research, 2001. Vol. 105. 106 p.
14. **Chelf B., Engler D., Hallem S.** How to write system-specific, static checkers in Metal. *SIGSOFT Softw. Eng. Notes*, 2002, no. 28, pp. 51–60. doi:10.1145/634636.586097
15. **Mossakowski T., Haxthausen A. E., Sannella D., Tarlecki A.** *CASL — the Common Algebraic Specification Language*. Logics of Specification Languages. Springer, Berlin, Heidelberg, 2008, pp. 241–298. doi:10.1007/978-3-540-74107-7_5
16. **Беляев М. В., Романенков Е. С., Игнатьев В. Н.** Моделирование библиотечных функций в промышленном статическом анализаторе кода. *Труды Института системного программирования РАН*, 2020, т. 32, № 3, с. 21–31. doi:10.15514/ISPRAS-2020-32(3)-2
17. **Itsykson V.** LibSL: language for specification of software libraries. *Softw. Eng*, 2018, vol. 9, pp. 209–220. doi:10.17587/prin.9.209-220
18. **Qiu D., Li B., Leung H.** Understanding the API usage in Java. *Information and Software Technology*, 2016, vol. 73, pp. 81–100. doi:10.1016/j.infsof.2016.01.011
19. **Astorga A., Srisakaokul S., Xiao X., Xie T.** PreInfer: Automatic inference of preconditions via symbolic analysis. *2018 48th Annual IEEE/IFIP Intern. Conf. on Dependable Systems and Networks (DSN)*, Luxem-

bourg, Luxembourg, 2018. IEEE, 2018, pp. 678–689. doi:10.1109/DSN.2018.00074

20. **Wen C., Cao J., Su J., Xu Z., Qin S., He M., Li H., Cheung S., Tian C.** *Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification*. Computer Aided Verification. CAV 2024. A., Gurfinkel, V. Ganesh (eds). Lecture Notes in Computer Science, Springer, Cham, 2024, vol. 14682, pp. 302–328. doi:10.1007/978-3-031-65630-9_16
21. **Ma L., Liu S., Li Y., Xie X., Bu L.** SpecGen: Automated generation of formal program specifications via large language models. arXiv preprint arXiv:2401.08807. 2024.
22. **Kogler P., Falkner A., Sperl S.** Reliable generation of formal specifications using large language models. *SE 2024-Companion*. Gesellschaft für Informatik eV, 2024, pp. 141–153. doi:10.18420/sw2024-ws_10

UDC 004.4'42, 004.4'418, 004.434
doi:10.31799/1684-8853-2024-4-24-39
EDN: RIUJGA

Modeling the behavior of standard library functions in program analysis

V. M. Itsykson[a], PhD, Tech., Professor, orcid.org/0000-0003-0276-4517, itsykson@yandex.ru
M. P. Onischuck[b], Post-Graduate Student, orcid.org/0000-0001-5359-0161
V. V. Kechin[a], Master Student, orcid.org/0009-0006-0845-2740
Y. A. Alekseev[c], Research Engineer, orcid.org/0009-0002-6925-6541
[a]ITMO University, 49, Kronverksky Pr., 197101, Saint-Petersburg, Russian Federation
[b]Peter the Great St. Petersburg Polytechnic University, 29, Politekhnicheskaia St., 195251, Saint-Petersburg, Russian Federation
[c]Coleman Tech LLC, 40, Mira Ave., room. 3/1, 129090, Moscow, Russian Federation

Introduction: The static analysis of software projects using external libraries is a difficult task, since it is associated with an exponential increase in the number of analyzed program traces, which leads to the need to artificially limit the analysis time or use simplifications leading to a deterioration in the completeness and accuracy of the analysis or to a decrease in test coverage. **Purpose:** To develop an approach for modeling the behavior of external libraries based on formal specifications approximating the behavior of the original functions, which makes it possible to simplify the analyzed projects and reduce the analysis time. **Methods:** Modeling the functions of the standard library using formal specifications in the LibSL language and subsequent automatic synthesis of simple equivalents of library functions based on formal descriptions. **Results:** We propose an approach to the specification of functions of the standard library of the Java language in the LibSL language, which replaces the most commonly used libraries and their functions with formal specifications that repeat the externally visible behavior of functions. The resulting approximating descriptions are translated into efficient replacements for the functions of the standard library and substituted for the analyzer instead of the original library functions. Experiments conducted on a part of the standard library of the Java language have shown the applicability of the approach: the analysis time of industrial-grade projects has been reduced by 30%. **Practical relevance:** The developed approach makes it possible to significantly expand the scope of static analysis in the tasks of defect detection and test generation, since due to the reduction of the state space, the accuracy and completeness of defect detection increases, and higher coverage of the program with generated tests is ensured.

Keywords — standard library, formal specifications, library behavior modeling, static program analysis, test generation.

For citation: Itsykson V. M., Onischuck M. P., Kechin V. V., Alekseev Y. A. Modeling the behavior of standard library functions in program analysis. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2024, no. 4, pp. 24–39 (In Russian). doi:10.31799/1684-8853-2024-4-24-39, EDN: RIUJGA

References

1. Wang G., Chattopadhyay S., Gotovchits I., Mitra T., Roychoudhury A. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2019, vol. 47, no. 11, pp. 2504–2519. doi:10.1109/TSE.2019.2953709
2. Rahaman S., Xiao Y., Afrose S., Shaon F., Tian K., Frantz M., Yao D., Kantarcioglu M. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. *Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security*, New York, NY, USA, 2019. Association for Computing Machiner, 2019, pp. 2455–2472. doi:10.1145/3319535.3345659
3. Madsen M., Livshits B., Fanning M. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2013. Association for Computing Machinery, 2013, pp. 499–509. doi:10.1145/2491411.2491417
4. Arzt S., Rasthofer S., Fritz C., Bodden E., Bartel A., Klein J., Le Traon Y., Octeau D., McDaniel P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 2014, vol. 49, no. 6, pp. 259–269. doi:10.1145/2666356.2594299
5. Akhin M., Belyaev M., Itsykson V. *Borealis Bounded Model Checker: The Coming of Age Story*. In: *Present and Ulterior Software Engineering*. M. Mazzara, B. Meyer (eds). Springer, Cham, 2017, pp. 119–137. doi:10.1007/978-3-319-67425-4_8
6. Delahaye M., Kosmatov N., Signoles J. Common specification language for static and dynamic analysis of C programs. *28th ACM Symp. on Applied Computing*, New York, NY, USA, 2013. Association for Computing Machinery, 2013, pp. 1230–1235. doi:10.1145/2480362.2480593
7. Krüger S., Spath J., Ali K., Bodden E., Mezini M. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, 2019, vol. 47, no. 11, pp. 2382–2400. doi:10.1109/TSE.2019.2948910
8. Ivanov D., Menshutin A., Fokin D., Kamenev Y. UTBot java at the SBST2022 tool competition. *Proc. of the 15th Work-*

shop on Search-Based Software Testing, Pittsburgh, PA, USA, 2022. IEEE, 2022, pp. 39–40. doi:10.1145/3526072.3527529
9. Hsieh C., Mitra S. Dione: A protocol verification system built with Dafny for i/o automata. *Integrated Formal Methods: 15th Intern. Conf.*, Bergen, Norway, 2–6 December 2019. Springer International Publishing, 2019, pp. 227–245. doi:10.1007/978-3-030-34968-4_13
10. Huisman M., Ahrendt W., Bruns D., Hentschel M. *Formal Specification with JML*. Karlsruhe, Karlsruhe Institute of Technology, 2014. 51 p.
11. Ouadjaout A., Miné A. A library modeling language for the static analysis of C programs. *International Static Analysis Symp.*, Springer, Cham, 2020, pp. 223–247. doi:10.1007/978-3-030-65474-0_11
12. Promsky A. V. C Program Verification: VC Explanation and the Standard Library. *Modeling and Analysis of Information Systems*, 2011, vol. 18, no. 4, pp. 157–167 (In Russian). doi:10.3103/S0146411612070127
13. Ball T., Rajamani S. K. *SLIC: A Specification Language for Interface Checking (of C)*. Technical Report MSR-TR-2001-21, Microsoft Research, 2001. Vol. 105. 106 p.
14. Chelf B., Engler D., Hallem S. How to write system-specific, static checkers in Metal. *SIGSOFT Softw. Eng. Notes*, 2002, no. 28, pp. 51–60. doi:10.1145/634636.586097
15. Mossakowski T., Haxthausen A.E., Sannella D., Tarlecki A. *CASL—the Common Algebraic Specification Language*. In: *Logics of Specification Languages*. Springer, Berlin, Heidelberg, 2008, pp. 241–298. doi:10.1007/978-3-540-74107-7_5
16. Belyaev M. V., Romanenkov E. S., Ignatyev V. N. Modeling of library functions in an industrial static code analyzer. *Proc. of the Institute for System Programming of the RAS (Proceedings of ISP RAS)*, 2020, vol. 32, no. 3, pp. 21–31 (In Russian). doi:10.15514/ISPRAS-2020-32(3)-2
17. Itsykson V. LibSL: language for specification of software libraries. *Softw. Eng*, 2018, vol. 9, pp. 209–220. doi:10.17587/prin.9.209-220
18. Qiu D., Li B., Leung H. Understanding the API usage in Java. *Information and Software Technology*, 2016, vol. 73, pp. 81–100. doi:10.1016/j.infsof.2016.01.011
19. Astorga A., Srisakaokul S., Xiao X., Xie T. PreInfer: Automatic inference of preconditions via symbolic analysis. *2018 48th Annual IEEE/IFIP Intern. Conf. on Dependable Systems and Networks (DSN)*, Luxembourg, Luxembourg, 2018. IEEE, 2018, pp. 678–689. doi:10.1109/DSN.2018.00074
20. Wen C., Cao J., Su J., Xu Z., Qin S., He M., Li H., Cheung S., Tian C. *Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification*. In: *Computer Aided Verification*. CAV 2024. A. Gurfinkel, V. Ganesh (eds). Lecture Notes in Computer Science, Springer, Cham, 2024, vol. 14682, pp. 302–328. doi:10.1007/978-3-031-65630-9_16
21. Ma L., Liu S., Li Y., Xie X., Bu L. SpecGen: Automated generation of formal program specifications via large language models. arXiv preprint arXiv:2401.08807. 2024.
22. Kogler P., Falkner A., Sperl S. Reliable generation of formal specifications using large language models. *SE 2024-Companion*. Gesellschaft für Informatik eV, 2024, pp. 141–153. doi:10.18420/sw2024-ws_10

УВАЖАЕМЫЕ АВТОРЫ!

Научные базы данных, включая Scopus и Web of Science, обрабатывают данные автоматически. С одной стороны, это ускоряет процесс обработки данных, с другой — различия в транслитерации ФИО, неточные данные о месте работы, области научного знания и т. д. приводят к тому, что в базах оказывается несколько авторских страниц для одного и того же человека. В результате для всех по отдельности считаются индексы цитирования, что снижает рейтинг ученого.

Для идентификации авторов в сетях Thomson Reuters проводит регистрацию с присвоением уникального индекса (ID) для каждого из авторов научных публикаций.

Процедура получения ID бесплатна и очень проста, есть возможность провести регистрацию на 12 языках, включая русский (чтобы выбрать язык, кликните на зеленое поле вверху справа на стартовой странице): https://orcid.org