МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий

Кафедра систем информатики

Направление подготовки: 230100 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ БАКАЛАВРСКАЯ РАБОТА

Разработка унифицированного интерфейса к решателям условий корректности

Черешнев Евгений Сергеевич

«К защите допущена»

Заведующий кафедрой,

д.ф.-м.н., профессор

Лаврентьев М. М. /……..

(фамилия , И., О.) / (подпись, МП)

«……»……20…г.

Научный руководитель

с.н.с., ИСИ СО РАН,

к.ф.-м.н., с.н.с.

Ануреев И. С. /…….

(фамилия , И., О.) / (подпись, МП)

«……»……20…г.

Дата защиты: «……»……20…г.

Автор: Черешнев Е. С. /……...

(фамилия , И., О.) / (подпись)

Новосибирск, 2014г.

МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий

Кафедра систем информатики

Направление подготовки: 230100 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

УТВЕРЖДАЮ

Зав. кафедрой Лаврентьев М.М.

…….

(подпись, МП)

«……»……20…г.

ЗАДАНИЕ

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ БАКАЛАВРСКУЮ РАБОТУ

Студенту *(ке)* Черешневу Евгению Сергеевичу

Тема: Разработка унифицированного интерфейса к решателям условий корректности

Исходные данные (или цель работы): Разработка унифицированного интерфейса к решателям условий корректности и его интеграция в качестве библиотеки языка Atoment.

Структурные части работы: Введение в предметно-ориентированные системы переходов и язык Atoment. Проектирование архитектуры библиотеки. Описание алгоритмов перевода элементов языка Atoment в выражения промежуточного языка.

Научный руководитель
с.н.с., ИСИ СО РАН
к.ф.-м.н., с.н.с
Ануреев И.С./……..
(фамилия , И., О.) / (подпись)
«…»……20…г.

Задание принял к исполнению
Черешнев Е.С./……..
(ФИО студента) / (подпись)
«…»……20…г.

Содержание

ВВЕДЕНИЕ

Недостаточная надежность характерна для современного программного обеспечения. Сложность программных систем является причиной многих человеческих ошибок при написании кода. В связи с этим стоимость разработки повышается. Как средство преодоления данной проблемы применяют тестирование, наиболее простой, очевидный и распространенный способ поиска ошибок. Однако тестирование имеет ряд недостатков, среди которых главным является невозможность доказать корректность проверяемого кода. Тем не менее появляются новые методы тестирования (Test-driven development, Fuzz testing), улучшаются старые, строятся целые методологии на основе накопленного опыта.

Верификация программного обеспечения с помощью формальных методов – перспективное направление для исследований. По сравнению с тестированием, формальные методы позволяют обосновать корректность программы и гарантировать, что алгоритм удовлетворяет абстрактной математической модели, которая может быть применена для написания кода, использующего этот алгоритм. Несмотря на такое преимущество, формальная верификация на данный момент не получила широкого распространения. В основном она находит применение в индустрии аппаратного обеспечения, где цена ошибки имеет большее коммерческое влияние. Это объясняется высокой стоимостью применения и существующими недостатками формальных методов.

Системы дедуктивной верификации в качестве одного из составляющих модулей содержат решатель условий корректности, который доказывает формулы, полученные в результате анализа исходного кода. Успешное доказательство условий корректности гарантирует соответствие программного кода математической модели, которая использовалась для их построения.

Актуальность работы: Доказательство условий корректности – узкое место в современных системах верификации программ. Для решения этой задачи такие системы используют разного рода средства автоматической поддержки доказательства (универсальные средства автоматического или интерактивного доказательства такие, как, например, PVS или HOL; SAT-решатели, SMT-решатели и др.). Большая часть существующих систем верификации интегрирована с одним или двумя решателями условий корректности. Поддержку большего числа решателей обеспечивает на настоящий момент только платформа верификации Why [4]. Поэтому создание унифицированного

интерфейса к решателям условий корректности, позволяющее расширить выразительную силу систем верификации, является актуальной задачей. Еще одним преимуществом унифицированного интерфейса является возможность комбинировать различные решатели условий корректности.

Цель работы: Разработать унифицированный интерфейс к решателям условий корректности и интегрировать его в качестве библиотеки языка Atoment.

Этапы работы:

1. Изучить область дедуктивной верификации программ и конкретные результаты, полученные в лаборатории теоретического программирования ИСИ СО РАН.
2. Изучить существующие программные средства дедуктивной верификации и автоматического доказательства.
3. Разработать язык интерфейса к решателям условий корректности в качестве подъязыка языка Atoment.
4. Разработать библиотеку, реализующую интерфейс. Библиотека должна включать интерпретатор языка интерфейса, а также модуль конвертации условий корректности из представления языка Atoment в представление конкретного решателя.
5. Разработать набор функциональных тестов для проверки корректности работы интерфейса.

Имеющийся фактический материал для работы:

1. Проводимое исследование использует следующую теоретическую базу: предметно-ориентированные системы переходов (унифицированный формализм для описания средств дедуктивной верификации программ) [2], язык выполнимых спецификаций предметно-ориентированных систем переходов Atoment [3], разработанные в лаборатории теоретического программирования ИСИ СО РАН.
2. Интерпретатор языка Atoment, система автоматической поддержки доказательства Microsoft Z3 [5].

1 Введение в предметно-ориентированные системы переходов и язык Atoment

ПОСП – системы переходов специального вида, предназначенные для определения предметно-ориентированных языков, используемых для разработки семантики языков программирования и проектирования, спецификации, прототипирования и верификации программных систем. ПОСП составляют основу комплексного подхода к решению указанных задач. Atoment – язык спецификации ПОСП.

В данном разделе описываются необходимые понятия и обозначения ПОСП, которые используются в следующих разделах для описании структуры разрабатываемой библиотеки, а также при описании реализуемых в этой библиотеке алгоритмов.

1.1 Предварительные понятия и обозначения

- A^* (A^+) – множество всех конечных (непустых) последовательностей, состоящих из элементов множества A.
- λ – пустая последовательность.
- $a_{\in X}$ обозначает элемент a множества X, аналогично вводятся обозначения $a_{\notin X}$ и $a_{\subseteq X}$.
- $pset(X)$ $(fpset\ (X))$ – множество всех подмножеств (конечных подмножеств) множества X.
- $X \to Y$ $(X \to_t Y)$ – множество всех (тотальных) функций из X в Y.
- $dom(f), range(f)$ – область определения и область значения функции f.

1.2 Описание объектов ПОСП

Элементы, сорта

El – множество объектов, называемых элементами.

Объект $() \in El$ называется пустым элементов.

$At \subseteq$ El – множество объектов, называемых атомами.

Функция $sSpec \in At \to pset(El)$ называется спецификацией сортов, если выполнены следующие свойства:

1. Атомы из множества $Srt = dom(sSpec)$ называются сортами. Множество $sSpec(a)$, где $a \in Srt$, называется доменом сорта a.
2. $* \in Srt$, и $sSpec(*) = El$. Сорт * называется наибольшим сортом.
3. $' * \in Srt$, и $sSpec(' *) = El$. Сорт '* называется квотированным сортом.
4. $() \notin sSpec(a)$ для любого $a \in Srt \setminus \{*, ' *\}$.
5. Если $a, b \in Srt \setminus \{*, ' *\}$, то $sSpec(a) \cap sSpec(b) = \emptyset$.

Подстановки

Пусть $Sub = At \to El^*$. Элементы множества Sub называются подстановками. Если $dom(\sigma) = \{b_1, \dots, b_n\}$, то подстановка σ может записываться как $((b_1, \sigma(b_1)), \dots, (b_n, \sigma(b_n))$. Функция подстановки $sub \in El^* \times Sub \to_t El^*$ относительно подстановки σ определяется следующим образом (применяется первое подходящее правило):

- $sub(\lambda, \sigma) = \lambda$
- $sub\big(u_{\in dom(\sigma)}, \sigma\big) = \sigma(u)$
- $sub\big(e_{\notin Exp}, \sigma\big) = e$
- $sub\big((p), \sigma\big) = (sub(p, \sigma))$
- $sub(e\ p, \sigma) = sub(e, \sigma)\ sub(p, \sigma)$, если $e \in El, p \in El^*$

Параметризованные имена

Последовательность элементов a называется спецификацией параметров имени, если a имеет вид $(b_1\ c_1) \dots (b_n\ c_n)$, $b_i \in At$ попарно различны и $c_i \in Srt$. Атом b_i называется i-м параметром a, сорт c_i – сортом b_i. Последовательность элементов $d\ par\ a$, где $d \in Exp$, называется спецификацией параметризованного имени, если b_i входят в d один раз. Она специфицирует параметрическое имя e вида $sub(d, \{\big(b_1, (!p\ \ c_1)\big), \dots, \big(b_n, (!p\ \ c_n)\big)\})$. Элемент d называется образцом e, b_i параметрами e. Число n называется местностью e и обозначается $arity(e)$. Элемент f называется экземпляром e относительно $g_1 \dots g_n \in El^*$, если $f = sub(d, \big((b_1,\ g_1),\ \dots\ ,\ (b_n,\ g_n)\big))$. Пусть $NSpec$ – множество всех спецификаций параметрических имен, Nam – множество всех параметризованных имен.

Переменные

Пусть $a \in NSpec, b \in Srt \{' *\}$. Элемент $(var\ a\ sort\ b)$ называется спецификацией параметризованной переменной. Он специфицирует параметризованную переменную c $\in Nam$ сорта b, если a специфицирует c. Пусть $VSpec$ – некоторое множество спецификаций переменных. Пусть Var – множество переменных, специфицируемых $VSpec$. Пусть $Var[b]$ – множество всех переменных из Var сорта b.

Состояния

Состояние s относительно базиса $(El, sSpec, VSpec)$ определяется как тотальная функция на Var, отображающая переменную $a \in Var[b]$ в элемент множества $sSpec(b_1) \times ... \times sSpec(b_n) \rightarrow_t sSpec(b) \cup \{()\}$, где $n = arity(a)$ и δ_i – сорт i-го параметра переменной a. Пусть St – множество всех состояний. Функция $s(a)$ называется значением переменной a в состоянии s.

Функции

Пусть $a \in NSpec, b \in Srt \setminus \{' *\}$. Элемент $(fun\ a\ sort\ b)$ называется спецификацией функции. Он специфицирует функцию $c \in Nam$ сорта b, если a специфицирует c. Элемент d называется вызовом функции c, если d – экземпляр имени a. Пусть $FSpec$ – некоторое множество спецификаций функций. Пусть Fun – множество функций, специфицируемых $FSpec$. Пусть $Fun[b]$ – множество всех функций из Fun сорта b. Означивание функций $funVal$ относительно базиса $(El, sSpec, FSpec)$ определяется как тотальная функция на $Fun \times St$, отображающая пару (c, s) в элемент множества $sSpec(b_1) \times ... \times sSpec(b_n) \rightarrow_t dmn(b)$, где $n = arity(c)$, δ_i – сорт i-го параметра функции c. Функция $funVal(c, s)$ называется значением функции c при означивании $funVal$ в состоянии s.

Значение элемента

Пусть $Var \cap Fun = \emptyset$, $UNam = Var \cup Fun$. Функция означивания элементов $val \in El \times St \rightarrow_t El$ относительно базиса $(El, sSpec, VSpec, FSpec, funVal)$, определяется следующим образом (применяется первое подходящее правило):

- $val(a,\ s) = e$, если $a \notin Exp$
- если $namSynMatch(a) = (b,\ c), namMatch(b, d) = e \in Var$, и $arity(e) = n$, то $val(a,\ s) = s(e)(d_1, \dots, d_n)$
- если $namSynMatch(a) = (b,\ c), namMatch(b, d) = e \in Fun$, и $arity(e) = n$, то $val(a, s) = funVal(e, s)(d_1, \dots, d_n)$
- $val(a, s) = ()$

Последовательность d определяется следующим образом (применяется первое подходящее правило):

- если ‘ $*$ – i-й сорт имен из b, то $d.i = c.i$
- $d.i = val(c.i,\ s)$

Элемент $val(a, s)$ называется значением элемента a в состоянии s.

Более детальное описание ПОСП можно найти в работе [1].

2 Краткий обзор библиотеки

Разрабатываемая библиотека расширяет язык Atoment путем добавления функциональности, отвечающей за решение задач доказательства, поиска контрпримеров, упрощения для условий корректности. Для этих целей в языке Atoment добавляются предопределенные элементы с фиксированной семантикой – команды. Команды взаимодействуют с контекстом доказательства, который включает условие корректности и спецификации используемых параметризованных имен.

Команды распознаются интерпретатором команд и обновляют контекст библиотеки (например, путем добавления посылок условия корректности) либо запускают процесс доказательства (либо процесс поиска модели, в которой условие корректности истинно). При добавлении определенной функциональности резервируется команда, отвечающая за запуск этой функциональности. Таким образом, библиотека предоставляет интерфейс в виде команд к функциональности сторонних решателей.

Для реализации было решено использовать решатель Microsoft Z3 с возможностью в дальнейшем расширить библиотеку другими решателями. Для обеспечения поддержки решателя необходимо реализовать адаптер, переводящий условия корректности из представления библиотеки в представление самого решателя.

Для реализации адаптеров решателей используется промежуточный язык, в котором присутствуют только необходимые конструкции для описания формул. Предварительно библиотека осуществляет преобразование условий корректности в промежуточный язык. Далее условия корректности в полученном представлении передаются адаптеру конкретного решателя.

При разработке промежуточного языка отдавалось предпочтение более простым конструкциям, что позволило упростить реализацию этапа предобработки условий корректности. Преобразования на данном этапе производятся в основном с выражениями промежуточного языка, поэтому реализационные сложности при написании алгоритмов, относящихся к предварительному анализу, напрямую зависят от структуры основных объектов, над которыми производятся преобразования: в первую очередь такими объектами описывается представление условий корректности на промежуточном языке.

Таким образом введение промежуточного языка обусловлено двумя причинами. Во-первых, это позволяет упростить написание адаптера для нового решателя: исключается

необходимость в изучении языка Atoment. Для реализации адаптера достаточно знать об API конечного решателя и промежуточного языка. Во-вторых, использование промежуточного языка на этапе предварительного анализа приводит к снижению сложности реализации части алгоритмов из-за более простого API промежуточного языка.

С другой стороны, введение промежуточного языка влечет за собой увеличение сложности получаемой на выходе формулы для доказательства, поскольку конкретный решатель может предоставлять конструкции, использование которых значительно упрощает формулы для самого решателя. Эти конструкции могли бы быть использованы напрямую при переводе условий корректности из языка Atoment. Данная проблема решается путем передачи адаптеру вместе с выражением также и исходного элемента, из которого было получено это выражение. Более подробно промежуточный язык рассматривается в 4.4.

Основной частью библиотеки является препроцессор (модуль предварительного анализа), который преобразует элементы языка Atoment, переданные библиотеке как аргументы команд (которые сами являются элементами), в формулы промежуточного языка. Язык Atoment определяет набор специфических конструкций, таких как наибольший и квотированный сорта, элементы, параметризованные имена. Данные конструкции должны быть преобразованы в формулы, которые могут быть переданы на вход решателю для проведения доказательства.

3 Формат команд

Библиотека поддерживает работу в двух режимах:

- Интерактивный. Команды считываются с консоли.
- Скриптовый. Файл с командами передается в качества аргумента запускаемой программе. Далее команды читаются из этого файла.

Командой является предопределенный элемент с фиксированной семантикой. Команды исполняются в пределах неявно определяемого контекста, который содержит формулировку задачи, к которой относятся вводимые команды.

Команды делятся на три группы в зависимости от своей функциональности:

- взаимодействие с контекстом (команды `namespec, premise, assert`);
- проверка истинности и выполнимости формул (команды `sat` и `prove`);
- комбинация решателей условий корректности (тактик) (команды `sat, prove, satOrSimplify, proveOrSimplify`).

Контекст включает условие корректности (которое для удобства можно вводить по частям, с помощью нескольких команд), может включать настройки, используемые в процессе доказательства (например, тактики), а также различные эвристики (например, вспомогательные ограничения, определения функций), которые могут упростить построение и доказательство формул, переведенных в формат соответствующих решателей.

Формальное описание контекста и операционной семантики команд приводится в разделе 6.

Далее описываются команды, реализованные в разработанной библиотеке:

1. `(namespec A)` – команда для добавления спецификации параметризованного имени в контекст. Если в дальнейшем, при задании условий корректности будет использоваться какая-либо параметризованная функция, то она должна быть предварительно объявлена с использованием данной команды. Элемент `A` специфицирует переменную или функцию: $A \in VSpec$ или $A \in NSpec$.
2. `(assert A)` – команда для задания утверждения: передаваемый в качестве аргумента элемент `A` должен иметь сорт `'*`. В дальнейшем при запуске

доказательства, решатель будет производить поиск модели, в которой утверждение принимает значение истина.

3. `(premise A)` – команда для задания посылки: передаваемый в качестве аргумента элемент должен иметь сорт `'*`. В дальнейшем при запуске доказательства, будет построена формула, в виде импликации, где посылкой является конъюнкция всех элементов, введенных в контекст с помощью данной команды.
4. `(prove A)` – команда для запуска процесса доказательства для переданного элемента сорта `'*`. Условие корректности будет сконструировано с использованием контекста, с учетом введённых ранее команд `premise` и `assert`. Для доказательства будет произведен поиск модели, в которой отрицание конечной формулы будет принимать значение истина. Отсутствие такой модели будет означать истинность формулы на любой модели.
5. `(sat A)` – команда для запуска процесса поиска модели, на которой формула, сконструированная по тем же правилам, что и для команды prove, будет принимать значение истина. Таким образом, данная команда эквивалентна команде `prove` с переданным отрицанием формулы.

После исполнения, команды второго типа выводят результат о статусе завершения и/или помещают возвращаемое значение в специальную переменную языка Atomete `val`. Например, команда `prove` выводит один из трех возможных результатов, которыми может завершиться процесс доказательства: формула тождественно истинна, формула не тождественно истинна либо результат отсутствует. Последнее означает, что решатель не имеет достаточных возможностей для успешного доказательства формулы. При возникновении подобной ситуации на уровне самой библиотеки можно реализовать обращение к другому решателю, который обеспечивает лучшую поддержку необходимой теории.

Строки, начинающиеся с символа «#» интерпретируются как комментарии. Одна команда может быть расположена на нескольких строках.

4 Архитектура библиотеки

Библиотека состоит из следующих модулей (рис 1):

- Интерпретатор команд
- Парсер команд
- Контекст языка Atoment
- Контекст промежуточного языка
- Модуль предварительной обработки (препроцессор) для преобразования из контекста языка Atoment в контекст промежуточного языка
- Адаптер – модуль, отвечающий за преобразование формул из контекста промежуточного языка в представление решателя

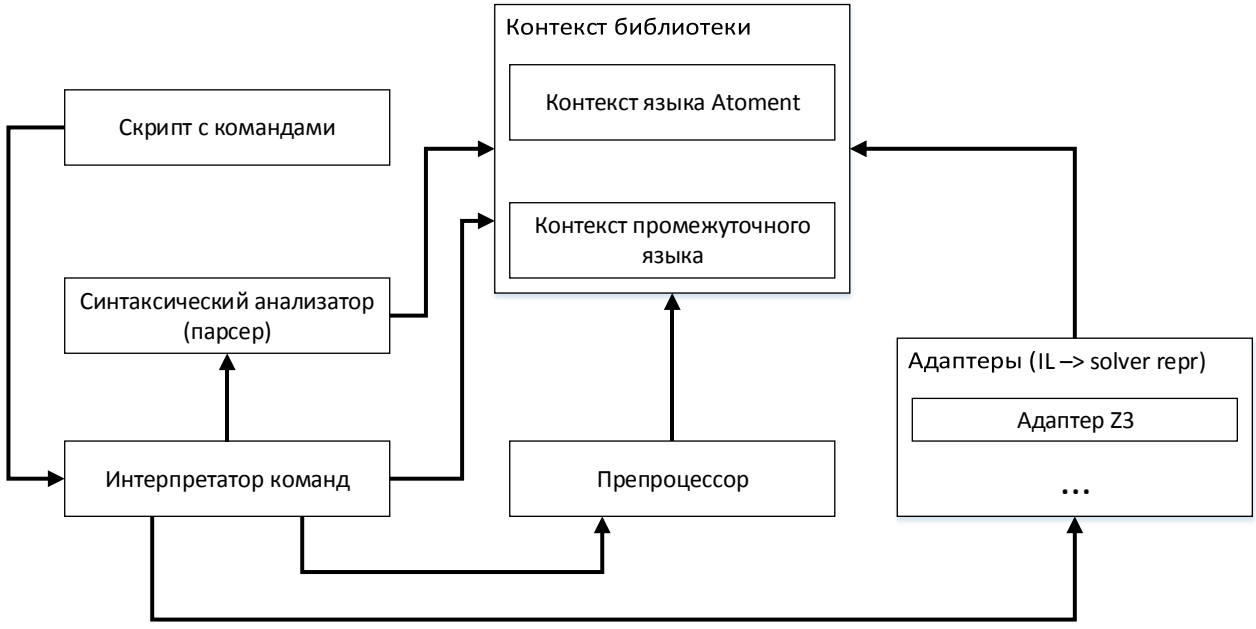

Рисунок 1. Схема компонентов библиотеки

4.1 Интерпретатор команд

Данный модуль отвечает за обработку передаваемых команд (либо посредством ввода с консоли, либо посредством чтения скрипта с командами из файла). Прочитанные команды передаются парсеру, который преобразует элементы языка Atoment в программные объекты библиотеки. Эти объекты хранятся внутри контекста, который затем передается интерпретатором другим модулям при обработке команд.

Для каждого типа команд определяется свой обработчик, которому передается преобразованный из строкового вида элемент. Обработчик обновляет контекст либо запускает доказательство условия корректности посредством вызова методов препроцессора.

В библиотеке понятие контекста используется для обозначения двух разных сущностей. Первое – это неявное состояние, которое обновляется командами (неявный контекст). Второе – это программный контекст: объект одного из классов `Context`, соответствующих либо языку Atoment, либо промежуточному языку Фактически контекст в смысле неявного состояния соответствует объединению двух программных контекстов, т.е. команда может менять состояние любого из программных контекстов.

4.2 Парсер команд

Парсер осуществляет разбор элементов языка Atoment в строковом виде и конструирует соответствующие элементы в объектной форме. Разбор осуществляется рекурсивно. При распознавании параметризованных имен происходит рекурсивный вызов функции для распознавания элементов-параметров. Распознавание параметризованных имен ведется на основе хранящихся в контексте языка Atoment спецификаций параметризованных имен.

Каждый раз после создания объекта, соответствующего распознаваемому элементу, происходит регистрация данного объекта в контексте, что позволяет в дальнейшем избежать создания объектов-копий при распознавании тех же элементов: при проверке они будут заменяться на объекты, которые уже находятся в контексте. Данный подход реализован с помощью программной идиомы PImpl (pointer to implementation) [6]: каждый объект-элемент содержит только указатель на настоящий объект, хранящийся в контексте.

Спецификации могут быть загружены при инициализации (для встроенных и вспомогательных функций, необходимых для разбора команд объявления спецификации). Другие спецификации должны быть объявлены до первой команды, использующей данную спецификацию.

4.3 Контекст языка Atoment

Все используемые объекты для описания элементов языка Atoment хранятся в программном контексте. К таким объектам относятся элементы всех типов: атомы, литералы, параметризованные имена, а также сорта и спецификации параметризованных имен. Контекст предоставляет методы, конструирующие объекты-элементы на основе переданных аргументов (имя, сорт, параметры). В случае, если в контексте уже хранится эквивалентный объект, то новый объект создан не будет, а метод возвратит уже существующий объект из контекста.

Контекст языка Atoment при инициализации создает объекты, описывающие встроенные сорта (`int`, `real`, `bool`, `atom`, наибольший сорт `*`, квотированный сорт `'*`), а также спецификации для встроенных параметризованных имен. Эти спецификации необходимы при синтаксическом анализе (на этапе работы парсера), поскольку парсер для определения подходящей спецификации при распознавании параметризованного имени производит поиск среди существующих спецификаций контекста.

Помимо служебных объектов, используемых на этапе распознавания и конструирования объектов-элементов, в контексте также хранятся составные части конечного элемента сорта `'*`, который в дальнейшем будет передан на доказательство. Для этого в контексте существует два списка с элементами сорта `'*`:

1. Посылки. Элементы, добавляемые командой `premise`.
2. Утверждения. Элементы, добавляемые командой `assert`.

При вызове команды `prove` или `sat` с передачей в качестве параметра элемента `E`, будет сконструирован элемент

$$(P_1\ and\ \ P_2\ and\ \ \dots and\ P_n) \rightarrow (A_1\ and\ A_2\ and \dots\ and\ A_m\ and\ E),$$

где P_i – элементы, введенные с помощью команды premise, а A_j – с помощью команды `assert`. Далее данный элемент используется препроцессором для преобразования

в формулу промежуточного языка. Вспомогательные команды `premise` и `assert` встроены в библиотеку для удобства ввода условий корректности.

4.4 Контекст промежуточного языка

Для обеспечения гибкости, необходимой для поддержки нескольких решателей, был введен промежуточный язык, который содержит более простые конструкции, в отличие от языка Atoment и позволяющий представить элементы языка Atoment в виде выражений этого промежуточного языка. Помимо этого исчезает необходимость непосредственной работы с API решателя при преобразовании элементов Atoment, что облегчает данный процесс ввиду наличия в языке Atoment специфических понятий, которые будет проще перевести в формулы базового промежуточного языка, а затем напрямую осуществить перевод в соответствующие объекты, предоставляемые API конкретного решателя.

Контекст промежуточного языка содержит объекты, соответствующие выражениям, в которые переводятся элементы из контекста языка Atoment. Контекст заполняется препроцессором на этапе преобразования элементов из языка Atoment. При этом в общем случае не существует однозначного сопоставления между элементами языка Atoment и формулами промежуточного языка, поскольку часть элементов могут быть упрощены, а остальные могут переводится в несколько независимых выражений.

В промежуточном языка выделяются следующие сущности:

- выражения (класс `Expr`);
- сорта (класс `Sort`);
- применения (класс `Application`);
- функции (класс `Fun`);
- значения (класс `Val`);
- переменные (класс `Var`).

Применения, значения и переменные относятся в выражениям. Любое выражение имеет фиксированный сорт. Применение описывается функцией и ее аргументами. Для любой функции фиксируются сорта параметров и их количество, а также возвращаемый сорт. Значения и переменные характеризуется сортом.

4.5 Модуль предварительной обработки для преобразования из контекста языка Atoment в контекст промежуточного языка (препроцессор)

Препроцессор – основной модуль библиотеки, отвечает за корректный и эффективный перевод элементов языка Atoment в промежуточный язык. В задачи данного модуля входит обработка и исключение специфических понятий языка Atoment, которые не имеют соответствующей поддержки в решателе. Например, язык Atoment позволяет оперировать с функциями возвращающими либо принимающими наибольший сорт, который может соответствовать любому другому сорту. В связи с этим появляется дополнительный этап, на котором происходит преобразование, заменяющее такие конструкции на конструкции, имеющие поддержку в используемых решателях.

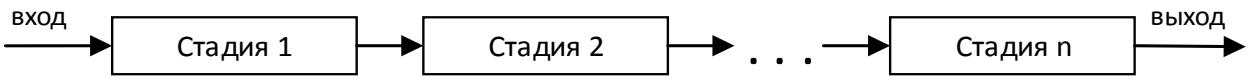

Рисунок 2. Паттерн pipes and filters

Для обеспечения гибкости и дальнейшей поддержки кода было решено использовать паттерн Pipes and filters (рис 2). Все основные преобразования, проводимые препроцессором разделяются на стадии, которые принимают на вход программные контексты (языка Atoment и промежуточного языка) и преобразуют их в соответствии с задачами самой стадии. Стадии регистрируются в препроцессоре. Далее контексты языка Atoment и промежуточного языка, поступая на вход препроцессору, по порядку преобразуются на каждой стадии. На выходе контекст промежуточного языка содержит результирующую формулу, которая может быть использована для передачи адаптеру какого-либо решателя. На данный момент в библиотеке реализованы следующие стадии (рис. 3):

- Стадия основного преобразования (класс `DirectConversionStep`). Элементы напрямую переводятся в формулы промежуточного языка: для каждого сорта языка Atoment создается соответствующий сорт в промежуточном языке (в том числе и для наибольшего и квотированных сортов), все параметризованные имена переводятся в применения, для каждой спецификации конструируется соответствующая ей функция и т.д.

- Стадия конструирования полной формулы на основе посылок и утверждений, введенных командами `premise` и `assert` (класс `AuxConversionStep`). На данной стадии конструируется результирующее выражение, которое составляется на основе введенных утверждений и посылок.
- Стадия исключения наибольшего сорта из формул (`CommonSortFixStep`). На данной стадии исходные формулы преобразуются так, что они больше не содержат наибольший сорт. Для этих целей конструируются новые неинтерпретированные сорта, или формула заменяется на несколько других формул, в каждой из которых используется конкретный сорт. Алгоритмы, использующиеся в данной стадии более подробно описываются в разделе 5.2.

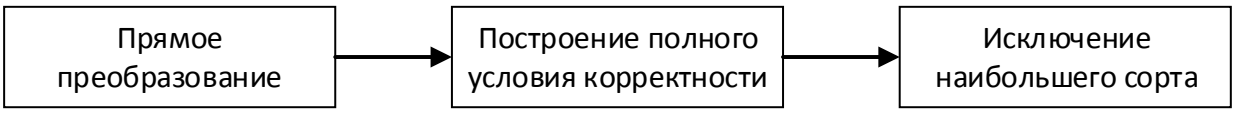

Рисунок 3. Стадии препроцессора

4.6 Адаптер – модуль, отвечающий за преобразование формул из контекста промежуточного языка в представление конечного решателя

Данный модуль отвечает за перевод формул из промежуточного языка в представление конечного решателя. Для перевода встроенных функций промежуточного языка в функции решателя используются таблицы соответствия. Эти таблицы конструируются динамически при запуске программы. Такие же таблицы создаются для встроенных сортов. Для остальных функций и сортов создаются неинтерпретированные функциональные символы и сорта при помощи API решателя.

Для обработки кванторов в адаптере для решателя Z3 предусмотрена отдельная обработка, поскольку в промежуточном языке кванторы относятся к функциям, а в решателе Z3 они имеют специальное представление.

Процесс запуска процесса доказательства в решателе Z3, также как и в других решателях, может приводить к зависанию. Данная проблема разрешается путем установки допустимого таймаута перед запуском.

Для добавления нового решателя в библиотеку необходимо реализовать интерфейс `IAdapter`, который содержит объявления двух методов: `trySat` и `tryProve`, которые запускают процесс поиска модели и процесс доказательства соответственно.

```
class IAdapter
{
public:
	virtual void trySat(const il::Expr& expr) = 0;
	virtual void tryProve(const il::Expr& expr) = 0;
};
```

5 Описание реализации основных алгоритмов библиотеки

5.1 Синтаксический анализ элементов языка Atoment

Элементы языка Atoment конструируются рекурсивно. Это означает, что при определении элемента могут использоваться элементы, построенные по таким же правилам, как и исходный элемент. При синтаксическом анализе предполагается, что любой элемент относится к одному из двух видов:

1) Атом или литерал (например '1', '3.14', 'true', 'abcd').
2) Выражение. Имеет вид $(e_1\ e_2 \ldots e_n)$, где $e_1, e_2, \ldots, e_n$ – элементы.

Такая структура позволяет реализовать алгоритм распознавания элементов достаточно просто. Основная логика заключена в методе, который осуществляет синтаксический разбор элемента языка Atoment. Этот метод в зависимости от вида элемента вызывает сам себя рекурсивно для распознавания внутренних элементов.

Каждая из команд библиотеки представляет собой один элемент языка Atoment, поэтому достаточно вызвать метод для распознавания с переданным строковым представлением элемента, который вернет объектное представление данного элемента.

Выражения могут являться параметризованными именами. В этом случае необходимо определить спецификацию путем сопоставления по сортам аргументов из списка всех зарегистрированных спецификаций.

5.2 Обработка наибольшего сорта в параметризованных именах

Язык Atoment предоставляет возможность использовать в спецификациях параметризованных имен наибольший сорт, что позволяет передавать в качестве аргументов элементы разных сортов в качестве параметров на одном и том же месте. Данный механизм не имеет прямой поддержки в решателе Z3, поэтому было предложено несколько подходов для преобразования формул, использующих параметризованные имена с параметрами наибольшего сорта.

Проблема, возникающая при обработке экземпляров параметризованного имени с параметрами наибольшего сорта заключается в том, что на стадии предобработки не всегда имеется возможность определить сорт элемента-параметра (например, если параметром

является вызов функции наибольшего сорта). Для решения данной проблемы можно применить два подхода:

- Исключение наибольшего сорта на этапе предварительного анализа. Наибольший сорт исключается из формул на этапе предобработки, в результате решатель получает формулы с конкретными сортами.
- Введение неинтерпретированного сорта, соответствующего наибольшему сорту со всеми необходимыми аксиомами.

В предложенной реализации используются оба подхода. Первый подход позволяет избежать усложнения путем анализа формул на этапе предобработки, тогда как второй позволяет преобразовывать любые формулы вне зависимости от наличия ограничений, позволяющих вывести конкретный сорт для каждого экземпляра параметризованного имени.

Далее приводятся более подробные описания подходов и их реализаций в составе библиотеки.

5.2.1 Исключение наибольшего сорта на этапе предварительного анализа

На стадии предобработки полученные формулы анализируются с целью однозначно определить сорта аргументов для всех вхождений экземпляров параметризованного имени. Также возможен случай, когда сорта аргументов определяются только при добавлении новых ограничений. Тогда можно провести преобразование исходных формул, используя условную функцию, которая будет производить выбор нужного объявления функции в зависимости от значений переменных.

Стоит отметить, что ограничения, от которых зависит сорт аргумента, могут передаваться пользователем в виде команд, либо определяться на основе встроенных эвристик (например, можно запустить процесс доказательства вспомогательной формулы, истинность которой будет гарантировать, что аргумент всегда имеет фиксированный сорт).

Пусть задана спецификация функции is_number сорта $Bool$, которая принимает единственный аргумент наибольшего сорта: $(fun\ (is\_number\ (!\ p\ *)\ sort\ bool)$. Для встроенных сортов `Int` и `Real` она возвращает элемент `true`, а для всех остальных сортов данная функция возвращает элемент `false`. Если в формулах встречаются только вызовы данной функции с фиксированными сортами, то достаточно объявить по

неинтерпретированной функции для каждого сорта и таким образом исключить наибольший сорт. Однако если в формулах встречается вызов этой функции с наибольшим сортом (когда параметр – вызов функции, возвращающей наибольший сорт), то необходимо провести дальнейший анализ для определения сорта параметра.

В случае, если информации для корректного вывода сорта параметра недостаточно, применяется второй подход, описываемый далее.

5.2.2 Введение неинтерпретированного сорта, соответствующего наибольшему сорту

Анализ, который осуществляет препроцессор, не может гарантированно вывести все сорта параметров. В этом случае вводится неинтерпретированный сорт с вспомогательными функциями, которые осуществляют перевод из этого сорта и в этот сорт для конкретных используемых в качестве сортов параметров.

Новый сорт фактически является аналогом наибольшего сорта, за исключением того, что все ограничения задаются только для ограниченного множества сортов, которые встречаются при вызове рассматриваемой функции. Дополнительно объявляются неинтерпретированные функции, которые отвечают за прямое и обратное преобразования для каждого из возможных сортов параметров параметризованного имени. Для данных функций вводятся аксиомы, гарантирующие однозначность таких преобразований, а также аксиомы, гарантирующие существование единственного значения конкретного сорта, соответствующее каждому значению наибольшего сорта.

Ниже приведен скрипт с соответствующими формулами для сортов `Int` и `Real` в формате SMT2 (S – наибольший сорт).

```
(declare-sort S)
(declare cast_int (Int) S)
(declare inv_cast_int (S) Int)
(declare cast_real (Real) S)
(declare inv_cast_real (S) Real)


(assert (forall ((x Int)) (= (cast_int (inv_cast_int x)) x)))
(assert (forall ((x Real)) (= (cast_real (inv_cast_real x)) x)))
(assert (forall ((x Int) (y Real)) (not (= inv_cast_int x) (= inv_cast_real x))))
```

При реализации данного подхода выяснилось, что у решателя Z3 существуют проблемы, связанные с поиском модели для выполнимой функции в том случае, если среди аксиом для наибольшего сорта встречаются сорта с бесконечным множеством значений, например, `Int`. Как возможное решение данной проблемы было предложено ограничивать бесконечный сорт с помощью введения нового конечного сорта, функций для преобразования и соответствующего набора аксиом. Ниже приводится скрипт с формулами в формате SMT2 с преобразованием для сорта `Int`, ограничивающее новый сорт `FiniteInt` ста элементами.

```
(declare-sort S)
(declare-sort Atom)
(declare-sort FiniteInt)
(declare-fun to (FiniteInt) Int)
(declare-fun from (Int) FiniteInt)

(assert (forall ((x FiniteInt)) (and (< (to x) 100) (>= (to x) 0))))
(assert (forall ((x FiniteInt)) (= (from (to x)) x)))
(assert (forall ((x Int)) (=> (and (< x 100) (>= x 0)) (= (to (from x)) x))))
```

5.3 Перевод предопределенных параметризованных имен в функции решателя

При распознавании элементов языка Atoment, которые являются параметризованными именами, необходимо определить соответствующую имени спецификацию. Поиск производится среди всех зарегистрированных в контексте спецификаций. При инициализации контекста языка Atoment осуществляется загрузка и регистрация встроенных спецификаций параметризованных имен (операции над встроенными сортами `int, real, bool`), а также вспомогательных спецификаций, используемых в модуле распознавания элементов при анализе команд. Таким образом, в случае, если команда содержит параметризованное имя, то соответствующая спецификация должна быть либо введена с помощью команды `namespec`, либо загружена из файла-скрипта со встроенными спецификациями.

На первой стадии препроцессора производится перевод каждой используемой спецификации в функцию промежуточного языка и каждого параметризованного имени в применение (Application). Перевод спецификаций осуществляется на основе заранее построенной таблицы с записями в следующем формате:

$$(e_1\ e_2 \dots e_m) \qquad [fun\_name\ sort_1\ \ sort_2 \dots sort_n],$$

где слева – параметризованное имя, справа – функция промежуточного языка, $e_i \in At$ или $e_i = (!\,p\ c_i)$ и $c_i \in Srt$, fun_name – имя соответствующей функции промежуточного языка, $sort_1, sort_2, \dots, sort_n$ – сорта параметров этой функции.

Далее, адаптер переводит все такие функции в функции конечного решателя на основе аналогичной таблицы. При разработке адаптера для решателя Z3, данная таблица была расположена в исходном коде, поскольку большинство из встроенных функций реализованы в API Z3 посредством механизма перегрузки операторов языка программирования C++.

6 Формальное описание языка обработки условий корректности

На языковом уровне унифицированный интерфейс к решателям условий корректности представляет собой предметно-ориентированный язык, предназначенный для решения задач, связанных с условиями корректности. В этом разделе дается формальное описание этого языка на базе языка Atoment, включающее его операционную семантику. Далее этот язык будем называть VCHL (Verification Condition Handling Language).

Любой предметно-ориентированный язык, создаваемый на базе языка Atoment, описывается четырьмя компонентами: множеством добавляемых сортов (расширяющих базовый набор сортов языка Atoment), множеством добавляемых переменных, описывающих состояние, множеством добавляемых функций и множеством добавляемых элементов. Добавляемые функции делятся на предопределенные и специфицируемые с помощью определений функций. Добавляемые элементы делятся на предопределенные и определяемые правилами переходов.

Множество добавляемых сортов языка VCHL включает сорта `validVC` и `satVC`. Значения сорта `satVC` возвращаются при проверке истинности формулы. Значения сорта `satVC` возвращаются при проверке выполнимости формулы.

Множество добавляемых функций языка VCHL включает функции со следующими спецификациями:

- `(fun (toEl x) par (x validVC) sort *)`,
- `(fun (toEl x) par (x satVC) sort *)`,
- `(fun (new validVC x) par (x *) sort validVC)`,
- `(fun (new satVC x) par (x *) sort satVC)`,
- `(fun (newq validVC x) par (x ‘*) sort validVC)`,
- `(fun (newq satVC x) par (x ‘*) sort satVC)`.

Первые две называются анализаторами, остальные – конструкторами сортов `validVC` и `satVC`, соответственно.

Если проверка истинности формулы A возвращает значение B, то связь `(toEl B)` с `A` определяется следующим образом:

- `(toEl B) = true` означает, что формула A истинна;
- `(toEl B) = false` означает, что формула `A` ложна;

- `(toEl B) = und` означает, что не удалось проверить истинность формулы `A`;
- `(toEl B) = (false C)` означает, что формула `A` ложна и элемент `C` является контрпримером. Конкретный вид контрпримера зависит от используемого решателя условий корректности;
- `(toEl B) = (simp C)` означает, что формула `A` истинна тогда и только тогда, когда формула `C` истинна. Формула `C` называется упрощением формулы `A`.

Если проверка выполнимости формулы `A` возвращает значение `B`, то связь `(toEl B)` с `A` определяется следующим образом:

- `(toEl B) = sat` означает, что формула `A` выполнима;
- `(toEl B) = unsat` означает, что формула `A` невыполнима;
- `(toEl B) = und` означает, что не удалось проверить выполнимость формулы `A`;
- `(toEl B) = (sat C)` означает, что формула `A` выполнима и элемент `C` является примером. Конкретный вид примера зависит от используемого решателя условий корректности;
- `(toEl B) = (simp C)` означает, что формула `A` выполнима тогда и только тогда, когда формула `C` выполнима. Формула `C` называется упрощением формулы `A`.

Функции `(new validVC (!p *))` и `(new satVC (!p *))` строят значения сортов `validVC` и `satVC` и обладают следующими свойствами:

- `(new validVC (toEl x)) = x`;
- `(new satVC (toEl x)) = x`;
- если не существует x сорта `validVC` такого, что `(toEl x) = y`, то `(new validVC (quote y)) = ()`;
- если не существует x сорта `satVC` такого, что `(toEl x) = y`, то `(new satVC (quote y)) = ()`.

Предопределенная функция `(quote (!p ‘*))` «замораживает» вычисление элемента и имеет следующую семантику: `val((quote A), s) = A` для любого элемента `A` и состояния `s`.

Функции `(newq validVC (!p ‘*))` и `(newq satVC (!p ‘*))` являются версиями функций `(new validVC (!p *))` и `(newq satVC (!p *))`, у которых аргумент не

вычисляется: `(newq validVC x) = (new validVC (quote x))` и `(newq satVC x) = (new satVC (quote x))`.

Множество добавляемых переменных языка VCHL включает следующие переменные:

1. Переменная `(sorts)` со спецификацией `(var (premiseList) sort exp)` хранит список используемых в условиях корректности сортов, в котором изначально присутствует встроенные сорта `(int, real, bool, atom, *, '*)`. Например, `val((sorts), s) = (A, B, C)`, где `Ai` – атомы, означает, что в состоянии s переменная (sorts) хранит список из сортов `A, B, C`.
2. Переменная `(namespecList)` со спецификацией `(var (namespecList) sort exp)` хранит список зарегистрированных спецификаций имен (переменных и функций), используемых в условиях корректности.
3. Переменная `(assertList)` со спецификацией `(var (assertList) sort exp)` хранит список утверждений (вводятся командой `assert`).
4. Переменная `(premiseList)` со спецификацией `(var (premiseList) sort exp)` хранит список посылок (вводятся командой `premise`).

Набор переменных языка VCHL может расширяться за счет переменных, специфичных для конкретных решателей условий корректности.

Опишем команды (добавляемые элементы) языка VCHL и их операционную семантику.

Команды `(namespec A), (assert A), (premise A)` описывают взаимодействие с контекстом доказательства, не зависящем от конкретных решателей условий корректности. Операционная семантика этих команд определяется следующим образом:

```
(if (namespec A) par A then (add A to list (namespecList)))

(if (assert A) par A then (add A to list (assertList)))

(if (premise A) par A then (add A to list (premiseList)))
```

Вспомогательный элемент `(add A to list B)` изменяет значение экземпляра B переменной сорта `exp` следующим образом: если `val(B, s) = (C)`, где `C` – последовательность элементов, то `val(B, s') = (C A)`, где `s, s'` – входное и выходное состояния, соответственно. Он имеет следующую семантику:

```
(if (add A to list B) par A B then (matchCases (!* B)
 (if (C) par (!s C) then (B ::= (C A)))
 (else (fail)) ))
```

Предопределенный элемент `(matchCases U W1 … Wn)` выполняет сопоставление элемента (или последовательности элементов) `U` с ветвями `W1, …, Wn`, а `(!* U)` означает, что сопоставляемым элементом является значение элемента U. Предопределенный элемент `(U ::= V)` означивает экземпляр `U` некоторой переменной значением элемента `V`. Подробнее об этих элементах см. [1].

Команды `(prove A with B)` и `(sat A with B)` описывает взаимодействие с конкретными решателями условий корректности на предмет проверки истинности и выполнимости формул, соответственно. Параметр B специфицирует конкретный решатель условий корректности и не влияет на обобщенную семантику этих команд, описанную ниже.

Операционная семантика команды `(prove A with B)` определяется следующим образом:

```
(if (prove A with B) par A B hvar W then
 (completeVC A) (W ::= (val))
 (modify
  (((((toEl (!n (val))) = true) => (!* W)) and
    (((toEl (!n (val))) = false) => (not (!* W))) and
    ((1 ofExp ((toEl (!n (val)))) = false) => (not (!* W))) and
    ((1 ofExp ((toEl (!n (val)))) = simp) =>
     ((2 ofExp ((toEl (!n (val)))) <=> (!* W)) )) or
   ((!n (val)) = und) )))
```

Предопределенный элемент `(modify U)` означает, что переход должен удовлетворять формуле `U`, связывающей входное и выходное состояние перехода. Элемент `(!n U)` обозначает значение элемента `U` в выходном состоянии. Подробнее об этих элементах и функциях см. [1]. Предопределенная функция `(A ofExp B)` возвращает `A`-й

по порядку элемент, образующий выражение B. Компонента hvar правила перехода специфицирует вспомогательные переменные (переменные истории), в которых хранятся промежуточные значения (элементы), появляющиеся при функционировании предметно-ориентированной системы переходов.

Элемент (completeVC A) дополняет проверяемое условие корректности A за счет формул, хранящихся в переменных (assumeList) и (assertList), и определяется следующим образом:

```
(if (completeVC A) par A then
 (matchCases (!* (assumeList)) (!* (assertList))
  (if (U) (V) par (!s U) (!s V) hvar WU hvar WV then
   (andList U) (WU ::= (val))
   (andList V A) (WV ::= (val))
   ((!* WU) => (!* WV))
```

Элемент (andList U) объединяет в конъюнкцию все элементы списка U и определяется следующими правилами перехода (правила упорядочены):

```
(if (andList) then ((val) ::= true))
(if (andList A) par A then ((val) ::= (quote A))
(if (andList A) par (!s A) then ((val) ::=q (and A))
```

Семантика предопределенного элемента (U ::=q V) отличается от семантики элемента (U ::= V) только тем, что элемент V не вычисляется (q означает quoted или квотированный элемент).

Операционная семантика команды (sat A with B) определяется следующим образом:

```
(if (sat A with B) par A B hvar W then
 (prove (not A) in B) (W ::= (val))
 (matchCases (!* (toEl W))
  (if (true) then ((val) ::= (newq satVC unsat)))
```

```
(if (false) then ((val) ::= (newq satVC sat)))
(if (false D) par D ((val) ::= (newq satVC (sat D)))
(if D par D ((val) ::= (newq satVC D))) ))
```

Третью группу команд языка VCHL образуют команды, комбинирующие работу решателей условий корректности (тактик). Поскольку аргументами таких команд являются тактики, будем называть эти команды тактикалами.

Язык Atoment достаточно выразительный, чтобы определять тактикалы с помощью правил перехода. Таким образом, набор тактикалов легко расширяется. Определим с помощью правил переходов несколько часто используемых на практике тактикалов.

Тактикал `(prove A withSeq B)` последовательно применяет к формуле A решатели из последовательности B до тех пор, пока не найдется решатель, который проверит эту формулу на истинность. Семантика этого тактикала определяется следующими правилами:

```
(if (prove A withSeq B) par A B then (prove A with B))
(if (prove A withSeq B C) par A B (!s C) then (prove A with B)
 (matchCases (!*(toEl (val)))
  (if und then (prove A withSeq C))
  (if (simp D) par D then (prove A withSeq C))
  (else) ))
```

Предопределенный элемент `(cases W1 … Wn)` выполняет проверку ветвей `W1, …, Wn`, до тех пор пока не встретиться ветвь с истинным условием. В этом случае выполняется компонента `then` этой ветви. Подробнее об этом элементе см. [1].

Тактикал `(sat A withSeq B)` последовательно применяет к формуле `A` решатели из последовательности B до тех пор, пока не найдется решатель, который проверит эту формулу на выполнимость. Семантика этого тактикала определяется следующими правилами:

```
(if (sat A withSeq B) par A B then (sat A with B))
(if (sat A withSeq B C) par A B (!s C) then (sat A with B)
```

```
(cases (!*(toEl (val)))
 (if und then (sat A withSeq C))
 (if (simp D) par D then (sat A withSeq C))
 (else) ))
```

Тактикал `(proveOrSimplify A withSeq B)` упрощает формулу `A`, последовательно применяя к ней решатели из последовательности B до тех пор, пока в результате очередного упрощения не получится проверенная на истинность формула. Семантика этого тактикала определяется следующими правилами:

```
(if (proveOrSimplify A withSeq B) par A B then (prove A with B))
(if (proveOrSimplify A withSeq B C) par A B (!s C) then
 (prove A with B)
 (matchCases (!* (toEl (val)))
  (if und then (prove A withSeq C))
  (if (simp D) par D then (proveOrSimplify D withSeq C))
  (else)))
```

Тактикал `(satOrSimplify A withSeq B)` упрощает формулу `A`, последовательно применяя к ней решатели из последовательности B до тех пор, пока в результате очередного упрощения не получится проверенная на выполнимость формула. Семантика этого тактикала определяется следующими правилами:

```
(if (satOrSimplify A withSeq B) par A B then (sat A with B))
(if (satOrSimplify A withSeq B C) par A B (!s C) then
 (sat A with B)
 (matchCases (!* (toEl (val)))
  (if und then (sat A withSeq C))
  (if (simp D) par D then (satOrSimplify D withSeq C))
  (else)))
```

7 Примеры доказательства условий корректности

В этом разделе унифицированный интерфейс используется для доказательства условий корректности для нескольких простых программ на языке C. Приведены примеры скриптов, реализующих доказательство этих условий корректности. Использование SMT-решателя Z3 позволило доказать все условия корректности для этих программ.

7.1 Вычисление корня монотонной функции

Функция вычисляет корень заданной монтонной функции f с точностью до eps на промежутке $[a; b]$. Известно, что $a < b, f(a) \leq 0, f(b) \geq 0, a \leq x \leq y \leq b \rightarrow f(x) \leq f(y)$.

Код функции на C:

```
double find_root(double a, double b, double eps)
{
	//{ P }
	double h = (b - a) / 2;
	double c = (a + b) / 2;
	while (h > eps && f(c) != 0)
	{
		//{ inv }
		h /= 2;
		if (f(c) > 0) c = c - h;
		else          c = c + h;
	}
	//{ Q }
	return c;
}
```

Аннотации для этой программы и порождаемые условия корректности взяты из [2].

Формулы для предусловия, инварианта цикла и постусловия функции имеют вид:

- P: $(a < b) \wedge (eps > 0) \wedge mon(a, b) \wedge nil(a, b)$
- inv: $(f(c) > 0 \rightarrow (a \leq c - h \wedge c \leq b)) \wedge (f(c) \leq 0 \rightarrow (c + h \leq b \wedge \mathrm{a} \leq c)) \wedge$
  $nil(c - h, c + h) \wedge mon(c - h, c + h) \wedge (h > 0)$
- Q: $nil(c - eps, c + eps)$

Условия корректности имеют вид:

- C1: $(inv \wedge \neg(h > eps \wedge f(c) \neq 0)) \rightarrow Q$
- C2: $(inv \wedge (h > eps \wedge f(c) \neq 0)) \rightarrow (f(c) > 0 \rightarrow inv\left(c \leftarrow c - \frac{h}{2}; h \leftarrow \frac{h}{2}\right))$
- C3: $(inv \wedge (h > eps \wedge f(c) \neq 0)) \rightarrow (f(c) \leq 0 \rightarrow inv\left(c \leftarrow c + \frac{h}{2}; h \leftarrow \frac{h}{2}\right))$
- C4: $P \rightarrow inv\left(c \leftarrow \frac{a+b}{2}; h \leftarrow \frac{(b-a)}{2}\right)$

Скрипт с командами для доказательства условий корректности:

```
(namespec (var (f (!p real)) sort real))
(namespec (var (mon (!p real) (!p real)) sort bool))
(namespec (var (nil (!p real) (!p real)) sort bool))

(namespec (var (inv (!p real) (!p real)) sort bool))

(namespec (var (a) sort real))
(namespec (var (b) sort real))
(namespec (var (c) sort real))
(namespec (var (h) sort real))
(namespec (var (eps) sort real))

#аксиомы для функций mon, nil
(premise (forall (x real) (y real) (((mon x y) and (x < y)) => ((f x) < (f y)))))
(premise (forall (x real) (y real) (z real)
              ((((mon x y) and (x <= z)) and (z <= y)) =>
                ((mon x z) and (mon z y)))))
(premise (forall (z real) (((f z) = 0.0) => (nil z z))))
(premise (forall (m real) (n real) (x real) (y real)
              (((((nil x y) and (m <= x)) and (x < y)) and (y <= n)) =>
                  (nil m n))))
(premise (forall (x real) (y real)
              ((mon x y) => ((nil x y) = (((f x) <= 0.0) and ((f y) >= 0.0))))))

(premise (a < b))
(premise ((eps < 1.0) and (eps > 0.0)))
(premise ((mon a b) and (nil a b)))

#функция inv
(premise (forall (c real) (h real) ((inv c h) =
       (((((((f c) > 0.0) => ((a <= (c - h)) and (c <= b))) and
       (((f c) <= 0.0) => (((c + h) <= b) and (a <= c)))) and
       (nil (c - h) (c + h))) and
       (h > 0.0)) and
       (mon (c - h) (c + h))))))

#C1
(prove (((inv c h) and (not ((h > eps) and (not ((f c) = 0.0))))) =>
        (nil (c - eps) (c + eps))))

#C2
(prove ((((inv c h) and (h > eps)) and (not ((f c) = 0.0))) =>
       (((f c) > 0.0) => (inv (c - (h / 2.0)) (h / 2.0)))))

#C3
(prove ((((inv c h) and (h > eps)) and (not ((f c) = 0.0))) =>
       (((f c) <= 0.0) => (inv (c + (h / 2.0)) (h / 2.0)))))

#C4
(prove (inv ((a + b) / 2.0) ((b - a) / 2.0)))
```

7.2 Обращение массива

Функция выполняет обращение первых $n+1$ элементов массива X чисел типа int длины $m+1$ и записывает результат в массив Z. При этом функция использует рекурсивный вызов для обращения массива меньшей длины.

Код функции на C++:

```
void rev(int Z[], int X[], int n)
{
	//{ P }
	if (n == 0)
		Z[0] = X[0];
	else
	{
		rev(Z, X, n - 1);
		for (int i = n; i != 0; i--)
		{
			//{ inv }
			Z[i] = Z[i - 1];
		}
		Z[0] = X[n];
	}
	//{ Q }
}
```

Аннотации для этой программы и порождаемые условия корректности взяты из [2].

Формулы для предусловия, инварианта, постусловия имеют вид:

- P: $((n \le m) \wedge (n \ge 0))$
- inv :$((i \ge 0) \wedge (i \le n) \wedge$
  $\forall k(((k \ge 0) \wedge (k \le i-1)) \to (Z[k] = X[n-1-k])) \wedge$
  $\forall k(((k \ge i+1) \wedge (k \le n)) \to (Z[k] = X[n-k])))$
- Q: $\forall k(\big((k \ge 0) \wedge (k \le n)\big) \to (Z[k] = X[n-k]))$

Условия корректности имеют вид:

- C1: $(P \wedge (n = 0) \wedge (Z[0] = X[0])) \to Q$
- C2: $(inv \wedge (i = 0)) \to ((Z[0] = X[n]) \to Q)$
- C3: $(n \ne 0) \to \forall Z(\forall k(\big((k \le n-1) \wedge (k \ge 0)\big) \to$
  $\big((Z[k] = X[n-1-k]) \to inv(i \leftarrow n)\big)))$
- C4: $(inv \wedge (i \ne 0) \wedge (Z[i] = Z[i-1])) \to inv(i \leftarrow i-1)$

Скрипт с командами для доказательства условий корректности:

```
(namespec (var (select (!p Array) (!p int)) sort int))
(namespec (var (store (!p Array) (!p int) (!p int)) sort Array))
(namespec (var (inv (!p int) (!p Array)) sort bool))

(namespec (var (Z) sort Array))
(namespec (var (X) sort Array))
(namespec (var (Q) sort bool))
(namespec (var (i) sort int))
(namespec (var (n) sort int))
(namespec (var (m) sort int))

#select-store аксиома для массивов
(premise (forall (a Array) (i int) (j int) (v int)
      (((i = j) => ((select (store a i v) j) = v)) and
      ((not (i = j)) => ((select (store a i v) j) = (select a j))))))

#функция для инварианта
(premise (forall (i int) (Z Array) ((inv i Z) =
      ((((i >= 0) and (i <= n)) and
      (forall (k int) (((k >= 0) and (k <= (i - 1))) =>
            ((select Z k) = (select X ((n - 1) - k)))))) and
      (forall (k int) (((k >= (i + 1)) and (k <= n)) =>
                        ((select Z k) = (select X (n - k)))))))))

#функция для постусловия
(premise (Q = (forall (k int) (((k <= n) and (k >= 0)) =>
                        ((select Z k) = (select X (n - k)))))))

#предусловие
(premise ((n <= m) and (n >= 0)))

#условия корректности
#C1
(prove (((n = 0) and ((select Z 0) = (select X 0))) => Q))

#C2
(prove (((inv i Z) and (i = 0)) => (((select Z 0) = (select X n)) => Q)))

#C3
(prove ((not (n = 0)) => (forall (Z Array)
      ((forall (k int) (((k <= (n - 1)) and (k >= 0)) =>
                  ((select Z k) = (select X ((n - 1) - k))))) => (inv n Z)))))

#C4
(prove ((((inv i Z) and (not (i = 0))) and
            ((select Z i) = (select Z (i - 1)))) => (inv (i - 1) Z)))
```

Заключение

В работе получены следующие результаты:

1. Описан язык интерфейса для решателей условий корректности в качестве расширения языка Atoment. Библиотека, реализующая интерфейс, предоставляет функциональность через команды – предопределенные элементы с заданной семантикой. Команды позволяют вводить условия корректности, запускать процесс поиска модели или процесс доказательства.
2. Разработан интерпретатор команд, обеспечивающий разбор и обработку команд. Подробно рассмотрена архитектура библиотеки. Приведены описания основных компонентов библиотеки: синтаксический анализатор, интерпретатор команд, препроцессор, адаптер. Описан промежуточный язык, упрощающий реализацию предварительной обработки условий корректности, а также написание адаптеров для решателей.
3. Представлены алгоритмы, используемые для преобразования элементов языка Atoment в представление, поддерживаемое конкретным решателем (модуль предварительного анализа). Описан алгоритм перевода предопределенных функций в функции, определяемые решателем.
4. Формально описан язык интерфейса для решателей условий корректности – VCHL. Описаны сорта, переменные, описывающие состояние и элементы, расширяющие язык Atoment, представлена операционная семантика команд,
5. Функциональность библиотеки проверена на наборе разработанных тестов. Использование SMT-решателя Z3 позволило доказать все условия корректности для программ из этих тестов.

Текущая реализация библиотеки поддерживает средство автоматического доказательства теорем Microsoft Z3. В работе рассмотрены подходы, используемые для исключения наибольшего сорта применительно к данному решателю.

Литература

1. Ануреев, И.С. Предметно-ориентированные системы переходов: объектная модель и язык / И.С. Ануреев // Системная информатика. - 2013. - № 1. - С. 1-34.
2. Непомнящий, В.А. Прикладные методы верификации программ / В.А. Непомнящий, О.М. Рякин. - М.: Радио-связь. - 1988. - 255 с.
3. Ануреев, И.С. Типовые примеры использования языка Atoment / И.С. Ануреев // Моделирование и анализ информационных систем. - 2011. - Т. 20. - № 4. - С. 7-20.
4. Why: a software verification platform: [Офиц. сайт]. URL: http://why.lri.fr/ (дата обращения: 28.05.2014).
5. Z3 - Home – CodePlex: [Офиц. сайт]. URL: http://z3.codeplex.com/ (дата обращения: 25.04.2014).
6. Эккель, Б. [Eckel B.], Философия C++. Введение в стандартный C++ / Пер. с англ. -СПб.: Питер. - 2004. - 572 с.