На пути к технологии разработки средств дедуктивной верификации программ*

Игорь Ануреев

Институт систем информатики имени А.П. Ершова
anureev@iis.nsk.su

Аннотация Статья представляет подход к разработке средств дедуктивной верификации программ. Основой подхода является предметно-ориентированный язык для данной предметной области. Сформулированы требования к такому языку и дано обоснование использования языка спецификации предметно-ориентированных систем переходов Atoment в качестве возможного кандидата. Особенности применения подхода и принципы, на которых он основан, проиллюстрированы на простых примерах. На базе подхода предполагается создать технологию разработки средств дедуктивной верификации программ.

Ключевые слова: дедуктивная верификация программ, предметно-ориентированный язык, предметно-ориентированные системы переходов, операционная семантика, аксиоматическая семантика, денотационная семантика, логика безопасности, язык Atoment

1 Введение

В статье предлагается предметно-ориентированный подход к разработке средств дедуктивной верификации программ (далее дедуктивных верификаторов). Термин «предметная ориентированность» применительно к подходу может иметь два значения. В первом случае, он означает ориентированность на предметную область, к которой применяется дедуктивная верификация. Во втором случае, он означает ориентированность на саму область разработки средств дедуктивной верификации программ. Мы рассматриваем предметную ориентированность во втором значении. Прежде чем описывать подход, определим необходимые термины этой предметной области и дадим классификацию существующих подходов к разработке средств дедуктивной верификации программ.

Дедуктивный верификатор применяется к аннотированной программе на некотором целевом языке программирования. Аннотированная программа на этом языке — это текст, построенный по определенным правилам

* Работа частично поддержана грантом РФФИ 11-01-00028-а и междисциплинарным интеграционным проектом СО РАН № 3 «Принципы построения онтологии на основе концептуализаций средствами логических дескриптивных языков».

из конструкций целевого языка и аннотаций. Аннотации описывают свойства программы на целевом языке. Аннотированная программа называется корректной, если эти свойства выполняются. Обычно в качестве языка аннотаций используется некоторый логический язык. Задача дедуктивного верификатора — доказать корректность аннотированной программы или указать условия, при которых выполнение этой программы некорректно.

Дедуктивный верификатор рассматривает аннотированную программу как формулу некоторого логического исчисления (называемого аксиоматической семантикой в контексте дедуктивной верификации) и преобразует ее в наборы формул некоторого логического языка в соответствии с определенной стратегией логического вывода в этом исчислении таким образом, что из истинности полученных в результате преобразования формул (называемых условиями корректности этой программы) следует корректность самой программы.

Для доказательства условий корректности дедуктивный верификатор применяет разного рода средства автоматической поддержки доказательства (универсальные средства автоматического или интерактивного доказательства такие, как, например, PVS или HOL; SAT-решатели, SMT-решатели и др.). Применительно к задаче дедуктивной верификации будем называть эти средства решателями условий корректности.

В качестве логического исчисления в дедуктивных верификаторах обычно используется логика Хоара или ее расширения. Инструменты, базирующиеся на логике отделимости (separation logic) и исчислении синонимов (alias calculus), также можно отнести к дедуктивным верификаторам в случае, если эти инструменты базируются на некотором виде аксиоматической семантики.

Опишем, как выглядит процесс разработки дедуктивного верификатора. Сначала разрабатывается концепт дедуктивного верификатора. Его разработка включает выбор целевого языка, языка аннотаций, аксиоматической семантики, методов оптимизации условий корректности, решателя условий корректности и т. д. После того, как он разработан, можно выделить три подхода к его реализации.

В первом подходе дедуктивный верификатор разрабатывается напрямую на некотором языке программирования общего назначения. Единственным достоинством этого подхода является большая степень свободы в оптимизации разрабатываемого верификатора. Эта степень ограничена только уровнем компетенций разработчика. Отметим недостатки подхода. Во-первых, при этом подходе высока сложность разработки, поскольку снижается уровень абстракции при переходе от концепта к реализации. Во-вторых, по той же причине, обоснование корректности верификатора представляет сложную проблему. В-третьих, верификатор не является гибким, поскольку ориентирован на конкретные целевой язык, язык аннотаций и аксиоматическую семантику.

Во втором подходе аннотированная программа транслируется в программу на промежуточном языке, для которого уже имеются дедуктивные ве-

рификаторы. Дополнительно к разработке транслятора из целевого языка в промежуточный язык, также разрабатываются средства обратной связи, которые интерпретируют результаты, полученные при верификации программы на промежуточном языке, в контексте исходной аннотированной программы. Как и в первом подходе, транслятор и средства обратной связи реализуются на некотором языке программирования общего назначения. Примерами промежуточных языков, ориентированных на дедуктивную верификацию, являются Boogie [1] и WhyML [2]. Этот подход имеет следующие преимущества. При его реализации знания конкретной аксиоматической семантики не требуется или требуется лишь в малой степени на уровне пользования дедуктивными верификаторами, имеющимися для промежуточного языка. Фактически разработка дедуктивного верификатора для целевого языка состоит в разработке транслятора и требует лишь знания операционной семантики целевого и промежуточного языков и (статической) семантики используемого языка аннотаций. Недостатки второго подхода проявляются прежде всего в случае, когда промежуточный язык сильно отличается от целевого языка. Во-первых, обоснование корректности трансляции становится сложной задачей. Во-вторых, усложняется разработка средств обратной связи. В-третьих, при трансляции теряется структурная и семантическая информация, которая могла бы оптимизировать генерацию условий корректности. Эта потеря, например, может быть результатом трансляции структур данных целевого языка в структуры данных промежуточного языка или результатом моделирования конструкций целевого языка конструкциями промежуточного языка. Помимо этого, разрабатываемый верификатор не является гибким, поскольку ориентирован на конкретные целевой язык, язык аннотаций и дедуктивные верификаторы, имеющиеся для промежуточного языка. Формат обратных связей, как правило, также не гибок.

В третьем подходе аннотированная программа транслируется в модель или теорию системы машинной поддержки доказательства (далее доказателя). К доказателям, используемым в дедуктивном подходе, относятся PVS, HOL, LCF2 и др. Можно выделить следующие преимущества этого подхода. Во-первых, доказатели, как правило, основаны на логиках высшего порядка, что позволяет использовать выразительные языки аннотаций в дедуктивных верификаторах. Во-вторых, доказатели используют продвинутые техники доказательства для формул этих логик. В-третьих, эти техники можно применять не только к доказательству условий корректности, но и при порождении условий корректности, таким образом оптимизируя их вывод. Отметим недостатки третьего подхода. Во-первых, доказатели сложны в освоении. Во-вторых, разработка модели или теории также сложна. В-третьих, доказательство условий корректности ограничено конкретным доказателем, в который вкладываются аннотированные программы. В-четвертых, возникает непростая проблема доказательства соответствия модели аннотированной программе. В-пятых, модель аннотированной программы, как правило, не использует дополнительную информацию, доступ-

ную при компиляции этой программы, так как учитывание этой информации усложнило бы модель и сделала бы ее непригодной на практике.

Предлагаемый нами подход нивелирует недостатки описанных подходов, сохраняя их преимущества, и может быть в дальнейшем развит в технологию разработки дедуктивных верификаторов. Он заключается в использовании предметно-ориентированного языка (далее ПОЯ) для разработки дедуктивных верификаторов.

2 Требования к предметно-ориентированному языку

Опишем требования, которые должны выполняться для ПОЯ. Набор этих требований не претендует на полноту и скорее является квинтэссенцией опыта, накопленного авторами в области дедуктивной верификации.

Требование 1. ПОЯ должен описывать объекты, которыми обычно оперирует дедуктивный верификатор. К ним относятся аннотированные программы, состояния программ, правила дедуктивного вывода (аксиоматической семантики) и стратегии их применения, условия корректности, интерфейсы к решателям условий корректности. Далее будем называть эти объекты объектами верификации.

Требование 2. Форматы описания объектов верификации на ПОЯ должны быть унифицированными для каждого типа объектов, т. е. не зависеть от конкретных используемых в дедуктивном верификаторе языков программ, аннотаций, правил и стратегий дедуктивного вывода, условий корректности и решателей условий корректности. Тем самым, обеспечиваются перечисленные выше преимущества второго и третьего подходов, обусловленные трансляцией. Будем называть результаты трансляции объектов верификации в ПОЯ представлениями или образами этих объектов.

Требование 3. Представления объектов верификации на ПОЯ должны быть структурно эквивалентными соответствующим им объектам. Структурная эквивалентность означает существование взаимно-однозначного отображения между объектами и их представлениями на уровне структуры объектов и, соответственно, структуры представлений. Тем самым нивелируются недостатки второго и третьего подходов, обусловленные трансляцией. Во-первых, упрощается разработка транслятора объектов конкретного языка в их представления на ПОЯ. Во-вторых, при такой трансляции не теряется информация. В-третьих, упрощается разработка средств обратной связи (перехода от представлений объектов к самим объектам).

Требование 4. Для текстовых объектов верификации (объектов, являющихся текстами) представления объектов на ПОЯ должны быть текстуально близкими к соответствующим им объектам. Текстуальная близость означает, что представления объектов по возможности должны выглядеть похожими на сами объекты. В частности, они должны использовать такие же ключевые слова и размещать их в том же самом порядке. В этом случае достигается читабельность и понимание представлений объектов, сравнимые с читабельностью и пониманием самих текстовых объектов. Это требо-

вание конфликтует с требованием 2, поэтому должен быть найден разумный компромисс.

Требование 5. Представления объектов верификации на ПОЯ должны быть концептуально близкими к соответствующим им объектам. Концептуальная близость означает, что представления используют ту же самую концептуальную структуру (понятия, атрибуты понятий, отношения между понятиями), что и объекты. Так семантические сущности, описывающие состояния программ на современных языках программирования, имеют сложную концептуальную структуру, которую важно точно отражать на уровне представлений соответствующих объектов.

Требование 6. ПОЯ должен описывать средства оперирования с представлениями объектов, сооветствующие средствам, которые дедуктивный верификатор использует для оперирования объектами верификации. Их можно разделить на базовые и дополнительные.

К базовым относятся средства, необходимые для выполнения дедуктивной верификации, к дополнительным — средства, которые оптимизируют этот процесс. Базовые средства включают средства взаимодействия с объектами через их представления и средства выполнения правил и стратегий дедуктивного вывода на представлениях объектов. Средства взаимодействия включают трансляторы аннотированных программ конкретных целевых языков в их представления на ПОЯ, средства построения обратных связей, средства взаимодействия с решателями условий корректности и т. д. Средства выполнения (интерпретации) правил и стратегий дедуктивного вывода, задаваемых в унифицированном формате на ПОЯ, позволяют реализовывать дедуктивные верификаторы, для которых эти правила и стратегии являются параметрами.

Дополнительные средства оперируют с представлениями объектов, и могут, например, включать средства комбинации дедуктивных верификаторов, средства трансформации аннотированных программ, средства комбинирования решателей условий корректности и средства трансформации условий коректности. Средства трансформации представлений целевых программ позволят комбинировать операционный и аксиоматический подходы к верификации программ. Предварительное преобразование представления (операционный подход) может упростить последующий вывод условий корректности в аксиоматической семантике. В частности, трансформации могут собирать информацию о программе, которая обычно собирается алгоритмами статического анализа или на этапе ее компиляции. Эта информация может быть использована для оптимизации вывода условий корректности. Так как решатели условий корректности — узкое место современных дедуктивных верификаторов, средства трансформации условий корректности повышают возможности таких верификаторов. Эти средства позволяют делать предобработку условий корректности и передавать решателю упрощенные условия корректности. При использовании нескольких решателей, они могут также распределять фрагменты условий корректности наиболее подходящим для их доказательства решателям. Кроме того, такие средства

могут быть использовано непосредственно для реализации простых решателей.

Требование 7. ПОЯ должен содержать развитый механизм сопоставления с образцом как на уровне синтаксической структуры (требования 3 и 4), так и на уровне концептуальной структуры (требование 5) представлений объектов. Сопоставление такого рода характерны для большинства средств оперирования представлениями объектов, перечисленных в требовании 6.

Требование 8. ПОЯ должен обеспечивать средства описания смысла конструкций целевого языка (операционной семантики целевого языка). Это требование необходимо для формального обоснования корректности разрабатываемого дедуктивного верификатора.

3 Использование языка Atoment в качестве ПОЯ

Язык Atoment [3] — результат наших исследований по автоматизации создания операционной и аксиоматической семантик языков программирования. В процессе его разработки возникла концепция предметно-ориентированного подхода к дедуктивной верификации и были сформулированы требования к ПОЯ для этой предметной области. Он не является идеальным кандидатом на роль ПОЯ, но обеспечивает компромиссные показатели по всем требованиям к ПОЯ и при отсутствии на данный момент других подходящих языков мы планируем использовать его для апробации этого подхода.

Язык Atoment обеспечивает описание объектов верификации (требование 1). Он использует два унифицированных формата для представлений объектов (требование 2), соответствующих текстовых и нетекстовым объектам.

Текстовые объекты представляются выражениями языка Atoment. Выражения — это унифицированная форма записи конструкций языка Atoment, подобная спискам языка Лисп. Выражения строятся из атомов с помощью специальных символов, к которым относятся пробельные символы и круглые скобки. Например, `ab` и `(ab (cd ef) gh)` — выражения. Атомом может быть любая последовательность Unicode символов, не содержащая специальных символов и символа кавычек (`"`). Специальный вид атомов — строки позволяют использовать в атоме специальные символы. Например, `ab` и `"ab (\"ss)"` — атомы. Требование 3 для текстовых объектов обеспечивается взаимно-однозначным соответствием между их структурой и структурой выражений, представляющих эти объекты. Рассмотрим, как обеспечивается выполнение требования 4 для текстовых объектов на примере представления конструкций целевого языка и языка условий корректности. Ключевые слова целевого языка представляются атомами. Для условного оператора `if A then B else C` ключевые слова представляются атомами `if`, `then` и `else`. Сам оператор представляется выражением `(if A′ then B′ else C′)`, где `A′`, `B′`, `C′` — представления конструкций `A`, `B`, `C` целевого языка. Условия корректности, как и любые другие формулы, также представляются выражениями. Так наиболее близким представлением формулы

$\forall x((a \leq x) \wedge (y \leq b))$ является выражение `(∀ x ((a ≤ x) ∧ (y ≤ b)))`. Обычно для удобства ввода вместо этого выражения используется выражение `(forall x ((a <= x) and (x <= b)))`.

Нетекстовые объекты представляются конечными наборами символов языка Atoment, а их (как правило, концептуальная) структура — состояниями в языке Atoment. Символы в языке Atoment — это выражения вида `(A)`, где `A` — последовательность атомов. Состояние в языке Atoment — это отображение, которое сопоставляет символам их интерпретацию (значение). Значением символа в состоянии являются функции заданной местности, сопоставляющие кортежу элементов элемент (Подобно тому, как атом является наименьшей синтаксической единицей языка Atoment, элемент является его наименьшей семантической единицей. Название Atoment является объединением двух слов — атом и элемент.). Местность этих функций совпадает с числом спецификаторов аргументов, входящих в символ. Спецификатор аргумента — это атом одного из видов `+v` или `-v`. Символы позволяют адекватно описывать концептуальную структуру (требование 5) семантических сущностей целевого языка. Например, переменная может быть представлена символами `(-v is variable)`, `(value of -v)` и `(type of -v)`. При таком представлении `s(-v is variable)(x) = true`, `s(type of -v)(x) = int` и `s(value of -v)(x) = 2` означает, что переменная с именем x имеет тип `int` и значение `2` в состоянии `s`.

Средства оперирования с представлениями объектов (требование 6), механизм сопоставления с образцом (требование 7) и средства описания операционной семантики целевого языка (требование 8) реализуются с помощью предметно-ориентированных правил перехода и контекстов их применения. Правило перехода — это выражение вида `(if A var B P then C)`, где A, B, C — последовательности выражений, называемые образцом, спецификацией переменных образца и телом правила, соответственно, необязательная последовательность выражений `P` обозначает дополнительные параметры правила. Правила перехода суть средства трансформации конфигураций. Конфигурация на языке Atoment — это пара, состоящая из последовательности выражений (далее программы на языке Atoment) и состояния. Правило `(if A var B P then C)` применимо к конфигурации `(D, s)`, если `D` имеет вид `E A′ F` и `A′` удовлетворяет образцу `A`, т. е. получается из `A` заменой переменных образца из `B` на некоторые выражения или последовательности выражений (значения переменных образца). Результатом применения правила будет конфигурация `(E C′ F, s′)`, где `C′` — результат замены в `C` переменных образца их значениями. Контекст применения определяет, как получается `s′`, может дополнительно модифицировать `C′` и накладывать дополнительные условие на применимость правила. Контекст определяется видом правил перехода (операционные, дедуктивные, условные, онтологические и т. д.). Так средства выполнения (интерпретации) правил и стратегий дедуктивного вывода применительно к некоторой аннотированной программе описываются дедуктивными правилами перехода, а операционная семантика целевых языков — операционными правилами перехода.

Рассмотрим пример применения правила. Операционное правило

```
(if (jump E) (break) var (+s E) then (jump E))
```

применимо к программе `(jump goto L) (break) (x := 3)` и преобразует ее в программу `(jump goto L) (x := 3)`. Состояние при этом не меняется. Спецификатор `+s` перед переменной образца `E`, означает, что ее значением может быть (возможно пустая) последовательность выражений. Если бы его не было, значением `E` могло бы быть только выражение.

Предметно-ориентированные системы переходов определяют операционную семантику выражений. Операционная семантика выражения `U` определяется множеством правил, образец которых включает выражение `U′` такое, что `U` получается из `U′` означиванием переменных образца. Таким образом, операционная семантика выражения определяется в контексте окружающих его выражений в программе на языке Atoment. Например, операционная семантика выражения `(break)` из примера выше определяется в контексте, когда этому выражению предшествует выражение `(jump E′)`, где `E′` — произвольная последовательность выражений.

Комбинирование дедуктивных верификаторов и решателей условий корректности осуществляется за счет предопределенных выражений языка Atoment, описывающих основные структуры управления (последовательное выполнение, разбор случаев, бэктрекинг и др.). Операционная семантика предопределенных выражений задается напрямую, а не определяется правилами перехода.

Для спецификации семантики формул (например, условий корректности) используется денотационная семантика выражений. Денотационная семантика выражений — это отображение, которое сопоставляет выражениям их значения, принадлежащие множеству элементов языка Atoment (денотату). Значением атома является сам атом. Значение выражения, отличного от атома, определяется семантикой символа, который является шаблоном для этого выражения. Так символ `(+v + +v)`, интерпретируемый как операция сложения чисел, является шаблоном для выражения `(2 + (3 + 4))` относительно сопоставленных аргументам выражений `2` и `(3 + 4)`. Спецификатор аргумента `+v` означает, что в качестве значения аргумента берется значение сопоставленного выражения. Таким образом, значением выражения `(2 + (3 + 4))` будет `9`. Спецификатор аргумента `-v` означает, что в качестве значения аргумента берется само выражение. Этот спецификатор используется, например, в символе `(forall -v -v)`, интерпретируемом как квантор всеобщности.

4 Применение предметно-ориентированного подхода

Рассмотрим применение предметно-ориентированного подхода на примере разработки дедуктивного верификатора для целевого языка, включающего простые типовые конструкции языков программирования. Из-за недостатка места мы ограничимся только вопросами разработки операционной семантики и логики безопасности (варианта аксиоматической семантики) для этих конструкций. Несмотря на свою простоту, пример иллюстрирует несколько

ключевых особенностей, лежащих в его основе: инкрементальность разработки, использование вспомогательных конструкций, гибкость разработки, разрешение на месте побочных эффектов при дедуктивном выводе. Будем разрабатывать верификатор сразу для выражений, представляющих конструкции целевого языка, считая, что требования структурной эквивалентности и текстуальной и концептуальной близости удовлетворены.

Рассмотрим оператор блока `({ A })`, где `A` — последовательность операторов. Заметим, что такое представление близко к представлению этого оператора на ряде императивных языков программирования. Опишем сначала его операционную семантику. Первую версию семантики зададим правилом

```
(if ({ A }) var (+s A) then A)
```

В соответствии с этой семантикой выполнение оператора `({ A })` сводится к выполнению последовательности операторов `A`.

Предположим теперь, что целевой язык содержит средства передачи управления (операторы посылки исключений, операторы `break`, `continue`, `return` и т. п.). В этом случае, такая семантика уже не подходит, поскольку оператор блока не должен выполняться, если перед ним произошла передача управления. Для описания этой ситуации введем понятие выражения перехода и концепцию просачивания выражения переходов. Выражение перехода имеет вид `(jump E)`, где `E` — последовательность выражений, которые кодируют информацию о том, какой оператор перехода сработал, и какую информацию он передал. Например, для оператора `(break)` соответствующее выражение перехода может иметь вид `(jump break)`. Зададим вторую версию семантики блока правилами

```
(if ({ A }) var (+s A) then A)

(if (jump E) ({ A }) var (+s E) (+s A) then A)
```

Дополнительное второе правило обеспечивает просачивание выражения перехода через оператор блока. Последовательность выражений `E` хранит информацию, которая передается при переходе. Таким образом, мы фактически «программируем» операционную семантику конструкций целевого языка на специализированном языке высокого уровня, уточняя ее на каждом шаге итерации. Как будет показано ниже, аналогичным образом «программируются» правила и стратегии дедуктивного вывода. В этом заключается инкрементальность разработки верификатора.

Эта версия, однако, не учитывает ситуацию, когда целевой язык включает операторы перехода по метке `(goto L)` и пустые помеченные операторы `(label L)`, оператор `(label L)` входит в `A` и в результате выполнения блока порождается выражение перехода `(jump goto L)`. В этом случае, выполнение должно быть продолжено с оператора `(label L)` последовательности `A`, а в соответствии с текущей семантикой блока выражение перехода `(jump goto L)` будет просачиваться дальше за блок. Третья версия семантики блока определяется правилами

```
(if ({ A }) var (+s A) then A (gotoStop A))

(if (jump E) ({ A }) var (+s E) (+s A) then A)
```

Вспомогательное выражение `(gotoStop A)` обрабатывает выражение перехода `(jump goto L)` в случае, когда оператор `(label L)` принадлежит `A`:

```
(if (gotoStop A) var A then)

(if (jump E) (gotoStop A) var (+s E) (+s A) then (matchCases (jump E)
  (if (jump goto L) var L then (matchCases (A)
   (if (B (label L) C) var (+s B) (+s C) then C (gotoStop A))
   (else (jump E))))
  (else (jump E))))
```

Предопределенное выражение `matchCases` выполняет разбор случаев в соответствии с образцом. Сначала оно формирует сопоставляемое выражение. В правиле выше это выражения `(jump E)` и `(A)` для первого и второго вхождения `matchCases`, соответственно. Затем для каждой ветви `(if U var V then W)` оно выполняет сопоставление сформированного выражения с образцом `U` для спецификации переменных образца `V` таким же образом, как для правила перехода. Первая подходящая ветвь выполняется таким же образом, как правило перехода. Если ни одна из ветвей не подошла, и `matchCases` содержит выражение `(else W)`, то выполняется последовательность `W`. В противном случае, с помощью механизма бектрекинга выполняется откат к предыдущей точке ветвления.

Эта версия операционной семантики для оператора блока иллюстрирует использование вспомогательных конструкций. Чтобы адаптировать правила операционной семантики, мы не меняем их, а добавляем вспомогательную конструкцию, которая специфицирует требуемую адаптацию, и для нее описываем операционную семантику. Соответствующие вспомогательные конструкции используются и при описании аксиоматической семантики.

Рассмотрим теперь, как с помощью правил перехода описываются правила и стратегии дедуктивного вывода. Особенностью нашего подхода является то, что вместо логики Хоара используется ее расширение — логика безопасности, предложенная в [4].

В логике Хоара ключевыми понятиями являются точки входа и выхода, которым соответствуют пред- и пост-условия. В ряде расширений логики Хоара допускается несколько точек входа и выхода.

В логике безопасности точек входа и выхода нет, и, соответственно, нет предусловия и постусловия. Вместо этого ряд точек программы снабжаются так называемыми условиями безопасности. Программа корректна в логике безопасности (безопасна), если при любом исполнении программы, если достигнута точка программы с условием безопасности, то это условие должно быть истинно. Таким образом, логику безопасности можно применять к программам без явных точек входа или выхода, например операционным системам или телекоммуникационным протоколам.

Важное свойство нашего подхода, являющееся одним из его преимуществ, заключается в том, что для многих конструкций целевого языка правила логики безопасности структурно не отличаются от правил операционной семантики. Таким образом, разработав операционную семантику

целевого языка, мы автоматически получаем большую часть правил логики безопасности этого языка. Также в силу идентичности правил логики безопасности и операционной семантики доказательство корректности правил логики безопасности становится более простой задачей.

Рассмотрим правила логики безопасности для блока, соответствующие последней третьей версии операционной семантики:

```
(if ({ A }) var (+s A) then A (gotoStop A))

(if (jump E) ({ A }) var (+s E) (+s A) then A)
```

Таким образом, правила логики безопасности и операционной семантики для блока совпадают. Однако логика безопасности для вспомогательного выражения `(gotoStop A)` меняется, поскольку в ней используется, как и в логике Хоара, инвариант метки:

```
(if (gotoStop A) var A then)

(if (jump E) (gotoStop A) var (+s E) (+s A) then (matchCases (jump E)
  (if (jump goto L) var L then (matchCases (A)
   (if (B (label L inv I) C) var (+s B) (+s C) I then (assert I))
   (else (jump E))))
  (else (jump E))))
```

Выражение `I` в помеченном операторе `(label L inv I)` соответствует инварианту метки `L`, т. е. `I` — условие безопасности, которое должно быть истинным при любом достижении этого оператора. Предопределенное выражение `(assert I)` добавляет условие безопасности `I` в соответствующую точку.

Побочные эффекты затрудняют применение логики Хоара. Например, чтобы применить классическое правило логики Хоара к условному оператору `(if A then B else C)`, требуется, чтобы условие `A` не имело побочных эффектов (не меняло состояние программы). В нашем подходе такой проблемы не возникает и условие `A` может иметь побочные эффекты, поскольку побочные эффекты не только в операционной семантике, но и в логике безопасности, разрешаются на месте.

Правила операционной семантики для условного оператора имеют вид:

```
(if (if A then B else C) var A (+s B) (+s C) then A (cases
 (if ((val) = true) then B)
 (else C)))

(if (jump E) (if A) var (+s E) (+s A) then (jump E))
```

Предопределенное выражение `cases` выполняет разбор случаев. Выполняется первая ветвь `(if U then V)`, для которой значение (денотационная семантика) выражения `U` равна `true`. Выполнение ветви заключается в выполнении последовательности выражений `V`. Если ни одна из ветвей не выполняется и `cases` содержит выражение `(else W)`, то выполняется последовательность `W`. В противном случае, с помощью механизма бектрекинга выполняется выполняется откат к предыдущей точке ветвления.

Как видно из первого правила, сначала вычисляется выражение `A`. Если при его вычислении происходят побочные эффекты, то они описываются правилами для соответствующих подвыражений выражения `A`. Результат вычисления выражения `A` сохраняется в нульместном символе `(val)`. Затем выполняется сравнение значения этого символа с `true` или `false`. Второе правило просачивает выражение перехода через условный оператор.

Правила логики безопасности для условного оператора совпадают с правилами операционной семантики. Однако операционная семантика ряда предопределенных выражений, например `cases`, отличается для контекстов применения, соответствующих операционным и дедуктивным правилам перехода [3].

5 Заключение

В заключение, ответим на вопрос, зачем разрабатывать универсальный поход к созданию верификаторов, когда существующие дедуктивные верификаторы для конкретных языков программирования нуждаются в существенном улучшении. Во-первых, предлагаемый подход является предметно-ориентированным, что означает, что он предоставляет средства, позволяющие в полной мере учитывать особенности целевых языков программирования и используемых методов и техник дедуктивной верификации. Во-вторых, технология быстрой предметно-ориентированной разработки дедуктивных верификаторов может оказаться востребованной, когда для целевого языка не существует средств дедуктивной верификации. В-третьих, существующие дедуктивные верификаторы представляют собой черные ящики, что не позволяет, например, улучшить реализованную аксиоматическую семантику или стратегию ее применения или скомбинировать операционный, аксиоматический и дедуктивный подходы при генерации условий корректности. В-четвертых, подход может быть применен для апробации новых методов дедуктивной верификации специалистами в этой области. В-пятых, применение подхода к специально разработанным учебным языкам программирования позволяет использовать его в сфере образования.

Список литературы

1. Barnett M., Chang B.-Y.E., DeLine R., Jacobs B., Leino K.R.M. Boogie: A modular reusable verifier for object-oriented programs // Proc. of Conf. "Formal Methods for Components and Objects Springer-Verlaf, Berlin, LNCS. 2006. Vol. 4111. P. 364–387.
2. Filliâtre J.-C., Paskevich A. Why3 – where programs meet provers // Proc. of the 22nd European Symposium on Programming, Springer-Verlaf, Berlin, LNCS. 2013. Vol. 7792. P. 125–128.
3. Ануреев И.С. Предметно-ориентированные системы переходов: объектная модель и язык // Системная информатика. 2013. № 1. С. 1–34.
4. Ануреев И.С. Дедуктивная верификация телекоммуникационных систем, представленных на языке Си // Моделирование и анализ информационных систем. 2012. Т. 19. № 6. С. 34–44.