МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Факультет **Информационных технологий и программирования**

Направление **Прикладная математика и информатика** Специализация : **2. Математическое и программное обеспечение вычислительных машин.**

Академическая степень **Магистр математики**

Кафедра **Компьютерных технологий** Группа **6538**

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Динамический вывод утверждений в языке программирования с поддержкой проектирования по контракту (на примере языка *Eiffel*)

Автор магистерской диссертации ___Поликарпова Н. И._____(подпись)
(Фамилия, И., О.)

Научный руководитель___Мейер Б._____(подпись)
(Фамилия, И., О.)

Руководитель магистерской программы_____(подпись)
(Фамилия, И., О.)

К з а щ и т е д о п у с т и т ь
Зав. кафедрой ____ВАСИЛЬЕВ В.Н. _____(подпись)
(Фамилия, И., О.)

“____”_____20 ____г.

Санкт-Петербург, 2008 г.

Содержание

Введение

Тематика настоящей работы связана с проблемами семантической корректности и спецификации программного обеспечения (ПО). Корректность бесспорно является одним из ключевых факторов качества ПО. В книге [1] она названа важнейшим качеством, так как «если система не делает того, что она должна делать, то все остальное — ее быстродействие, хороший пользовательский интерфейс — не имеет особого значения» [1, глава 1]. На практике же корректности программ долгое время уделяли слишком мало внимания. В результате вопиющая некорректность ПО не только стала привычным делом для пользователей, но и, можно сказать, вошла в поговорку[1]. В последние годы проблема обеспечения корректности является одной из наиболее актуальных в программной инженерии. Об этом говорит большое число исследований и разработок в области тестирования, доказательства программ, проверки моделей (model checking).

Неформально программа корректна, если она делает то, что от нее ожидает пользователь. Чтобы говорить о корректности формально, необходимо ввести понятие спецификации. В этом случае *корректность* — это способность программной системы выполнять свои задачи так, как они определены ее спецификацией. Таким образом, программа не может быть корректна или некорректна «сама по себе», а только по отношению к заданной спецификации.

Одна из главных преград на пути к обеспечению корректности ПО — то, что у большинства программных систем вообще нет спецификации или она настолько неформальна, что ее согласованность с реализацией не может быть однозначно установлена. Заказчик ПО обычно не способен создать формальную спецификацию, так как не владеет необходимым математическим аппаратом. Программисты не пишут спецификации, так как не видят в этом прямой выгоды[2]: создание формальной спецификации зачастую требует усилий, как

[1] Вспомните одну из многочисленных шуток на эту тему «Если бы программисты строили дома». Вот характерная цитата из нее: «Оказалось хуже: мы забыли про фундамент. В проекте он, конечно, описан, но ведь документацию читают только ламеры».

[2] Складывается классическая ситуация «Верхи не могут, низы не хотят». Очевидно в сфере спецификации ПО назревает революция!

минимум, сопоставимых с написанием исполняемого кода, в то же время от нее нет мгновенного эффекта, как, например, от реализации новой функции программной системы (особенно сложно объяснить необходимость спецификации заказчику). Несмотря на то, что в настоящее время существует множество различных языков и методов формальной спецификации (*Z* [2], *B* [3], *Larch* [4], *JML* [5]), большинство из них применяется лишь в академических целях, но не в практическом программировании.

Один из практичных методов формальной спецификации ПО — *проектирование по контракту* [6]. Поддержка этого метода встроена в такие языки программирования, как *Eiffel*, *D*, *Oxygene*, *Lisaac*, *Sather* и другие. Далее в настоящей работе речь о проектировании по контракту будет идти только в контексте языка и метода программирования *Eiffel* [1, 7].

При проектировании по контракту процесс спецификации программной системы неотделим от ее реализации, так как спецификации, называемые *контрактами*, являются частью языка программирования.

- Каждый класс может содержать *инвариант класса* — набор булевых выражений (*предложений*), каждое из которых должно выполняться во всех стабильных состояниях экземпляров данного класса. Стабильными называются те состояния объекта, которые могут наблюдаться его клиентами [1, глава 11]. Таким образом, инвариант класса должен устанавливаться его процедурами создания и поддерживаться всеми экспортированными подпрограммами.

- Каждый компонент класса можно снабдить *предусловием* и *постусловием* (они также состоят из отдельных предложений). Любое выполнение компонента, начавшееся в состоянии, в котором каждое предложение предусловия было истинно, обязано завершиться, причем в таком состоянии, что каждое предложение постусловия будет истинно. В постусловии могут использоваться *old-выражения*, значения которых вычисляются в начальном состоянии (до выполнения компонента). Таким образом, постусловие устанавливает отношение между начальным и конечным состоянием объектов.

- В тело подпрограммы между любыми двумя инструкциями[3] может быть внедрена инструкция *проверки*, содержащая набор предложений, каждое из которых должно быть истинно в момент выполнения данной инструкции.

- Для спецификации циклов существуют *инварианты циклов* и *варианты циклов*. Инвариант цикла — это набор предложений, который должен быть истинным в начале цикла (после выполнения составной инструкции `from`) и после каждой итерации цикла (после каждого выполнения составной инструкции `loop`). Вариант цикла — это единственная составляющая контрактов, не являющаяся утверждением. Вариант представляет собой целочисленное выражение, значение которого должно быть неотрицательным после каждой итерации цикла, причем с каждой итерацией оно должно уменьшаться.

Спецификации *Eiffel* называются контрактами, так как они построены на жизненной метафоре контракта или договора между поставщиком услуг и его клиентом. Как и в мире людей, в мире ПО у каждой из сторон есть свои обязательства, и если клиент не выполнил свою часть договора (предусловие), то поставщик вовсе не обязан выполнять свою (постусловие). В мире ПО контракт заключается между программными модулями, то есть в объектно-ориентированном случае — между классами.

Что делает проектирование по контракту практичным?

Контракты составляются на языке программирования. От программиста не требуется учить еще один язык, разбираться в кванторах и логике предикатов. Все, что нужно знать — это обычные выражения *Eiffel*.

Контракты могут быть частичными. Создание полных формальных спецификаций требует много времени и усилий. Но контракты не обязательно должны быть полными. Часто какое-то свойство класса само собой приходит на ум во время работы над реализацией и хочется куда-то записать его, чтобы не забыть. И контракт — это самое подходящее место. В этом случае создание спецификации практически не требует усилий. Конечно, в

[3] Здесь и далее «между любыми двумя инструкциями» означает также перед первой и после последней инструкии.

таких условиях спецификации получаются довольно слабыми: например, их нельзя использовать для целей статического анализа, но для многих других целей они вполне пригодны.

Контракты — часть стандарта языка. Разработчики часто не доверяют сторонним нотациям, инструментам и библиотекам. Поддержка контрактов встроена в *Eiffel*, является частью стандарта языка и поддерживается всеми основными компиляторами и средами разработки. Снабдив код на *Eiffel* контрактом, разработчик может быть уверен, что этот контракт поймет любой другой программист на *Eiffel*, а тажке любой инструмент, работающий с языком.

От контрактов есть явная и мгновенная польза. Это не спецификация ради спецификации. Основные среды разработки *Eiffel* поддерживают проверку контрактов во время выполнения, отладка становится гораздо проще. Кроме того, именно контракты используются в качестве основной документации при повторном использовании чужого кода.

К сожалению, не все разработки ПО ведутся на *Eiffel*. Бóльшая часть существующих программных систем лишена даже таких, частичных спецификаций. Возможный выход из положения состоит в следующем: если спецификации необходимы, но люди их не создают, то пусть их создает машина. Такова основная идея *динамического вывода утверждений* [8] — автоматического выявления свойств программы из наблюдений, сделанных во время ее выполнения. Этот подход был предложен в 1999 г. *М. Эрнстом* и его коллегами из университета Вашингтона и с тех пор успешно применяется для программ, написанных на *Java*, *C/C++*, *Perl* и других языках. Первая реализация этого подхода — детектор утверждений *Daikon* [9], созданный той же группой ученых, одновременно является, насколько известно автору, наиболее успешным и широко используемым инструментом динамического вывода утверждений.

Цель настоящей работы — объединить упомянутые подходы к спецификации ПО и выяснить, какие преимущества может дать их сочетание. Нужен ли в *Eiffel* динамический вывод утверждений и для каких целей? Для того, чтобы ответить на этот вопрос, необходимо решить следующие задачи.

1. Для начала, необходимо понять, каким должен быть вывод утверждений в чисто объектно-ориентированном языке, таком как *Eiffel*. Если резуль-

таты вывода утверждений в программах на *Java* записываются в виде комментариев и предназначены, в основном, для чтения человеком, то в *Eiffel* выведенные утверждения станут частью формального текста программы и будут обрабатываться компилятором и другими автоматическими средствами. Поэтому уровень требований к выводу утверждений в *Eiffel* значительно выше. Кроме того, философия и правила языка *Eiffel* сильно отличаются от гибридных языков семейства *C*, на которых преимущественно апробировалась техника вывода утверждений.

2. Далее необходимо реализовать *Eiffel*-интерфейс к детектору утверждений *Daikon* (это позволит *Daikon* работать с программными системами написанными на *Eiffel*).

3. Наконец, необходимо провести эксперимент по применению детектора *Daikon* к различным классам на *Eiffel* и оценить практическую пользу от вывода утверждений.

Дальнейшее изложение имеет следующую структуру. Глава 1 содержит описание текущих достижений в области динамического вывода утверждений. Глава 2 посвящена особенностям вывода утверждений в языке *Eiffel*. В главе 3 описано инструментальное средство *CITADEL* — реализация *Eiffel*-интерфейса к детектору *Daikon*. Наконец, глава 4 содержит описание проведенного эксперимента.

Глава 1.

Состояние вопроса

Техника динамического вывода утверждений и инструментальное средство *Daikon* кратко описаны в работе [10] и более подробно в работах [8, 11, 12].

Daikon выводит свойства программы из истории значений некоторого набора *переменных*[1] в определенных *точках программы*. Обычно точки программы, которые интересуют пользователя — это точки входа и выхода из подпрограмм. В роли переменных выступают различные выражения, которые имеют смысл в данной точке программы, например: текущий объект, формальные аргументы подпрограммы, результат функции, атрибуты текущего объекта и других переменных.

Программная система, выполняющая динамический вывод утверждений, состоит из нескольких компонентов (рис. 1).

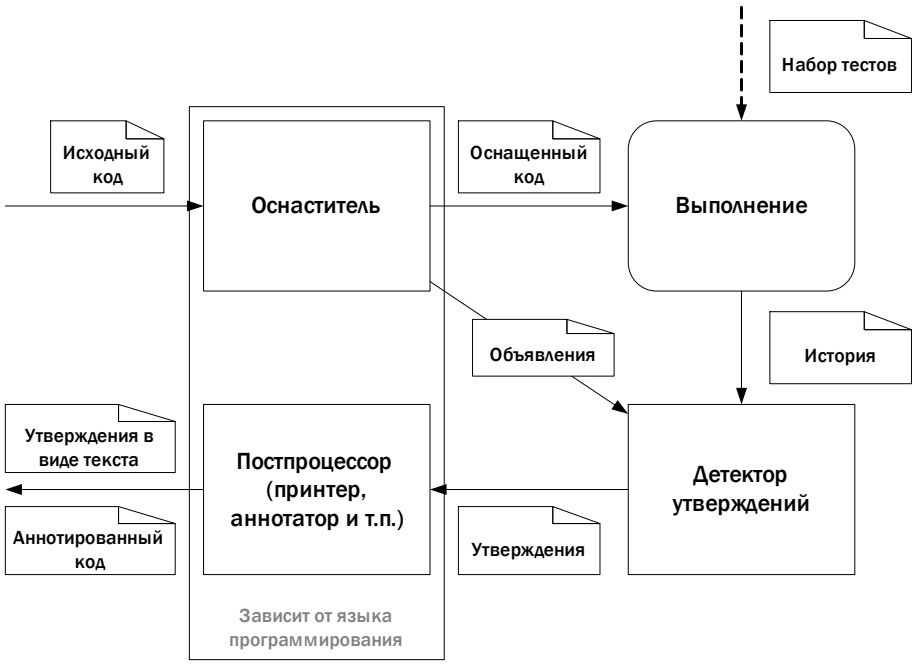

Рис. 1. Система динамического вывода утверждений

[1]Термин «переменная» используется в литературе по динамическому выводу утверждений и в настоящей работе в более широком смысле, чем в большинстве языков программирования, и, скорее, соответствует понятию «выражение».

Сначала *оснастительъ* обрабатывает исходный код программы, формируя *объявления точек программы* (статическую информацию об интересных точках программы и переменных) и внедряя в интересные точки программы инструкции для записи значений переменных в файл *истории*. В результате получается оснащенный код, который затем запускается либо сам по себе, либо на некотором наборе тестов. Оснащенная программа во время выполнения создает файл истории. Этот файл вместе с файлом объявлений подается на вход *детектору утверждений*. Выведенные им утверждения можно в дальнейшем обрабатывать различными *постпроцессорами*: печатать в подходящем формате, вставлять как аннотации в исходный код. Как показывает опыт, последнее наиболее полезно, поэтому в дальнейшем из всех постпроцессоров будем рассматривать только *аннотатор*.

Из перечисленных компонентов только оснастительь и аннотатор зависят от языка, на котором написана исходная система. Эти два компонента формируют интерфейс, позволяющий универсальному детектору утверждений работать с программами на разных языках (а также с другими данным, необязательно полученными в результате выполнения программы). В литературе название *Daikon* употребляется и по отношению к системе вывода утверждений в целом, и по отношению к детектору в частности. В настоящей работе эти два понятия называются соответственно *система Daikon* и *детектор Daikon*.

Принцип работы детектора *Daikon* очень прост: он пробует все имеющиеся шаблоны утверждений на всех подходящих переменных. Если какое-то утверждение не нарушается ни на одним наблюдении в данной точке программы, то *Daikon* считает его верным в этой точке и сообщает о нем. Конечно, это лишь основная идея: в детекторе используется целый ряд эвристик, позволяющих сократить число нерелевантных выведенных утверждений и повысить быстродействие. Они описаны в работе [8].

Как указано в работе [10], утверждения, выведенные системой *Daikon*, применялись, среди прочего, для генерации тестовых наборов, предсказания несовместимости при интеграции компонентов, автоматизации доказательств, исправления противоречивых структур данных и проверки корректности потоков данных.

Важно отметить, что лишь часть выведенных утверждений обычно являются свойствами анализируемого кода, остальные же отражают специфику

условий, в которых исполнялась оснащенная система (свойства тестового набора, входных данных, окружения). С одной стороны, это один из главных недостатков динамического вывода утверждений, так как его первоначальной целью является вывод спецификации программы. С другой стороны это открывает новые возможности применения утверждений, выведенных динамически: они могут использоваться для оценки тестовых наборов и как своего рода высокоуровневая отладочная информация. Кроме исходной системы и входных данных результат вывода утверждений зависит также от интерфейса (какие переменные он обнаруживает в тех или иных точках программы) и от множества шаблонов утверждений (*грамматики утверждений*) в детекторе. Утверждения, невыразимые в этой грамматике, не могут быть выведены. Таким образом, в выведенной спецификации в общем случае недостает некоторых утверждений истинной, искомой спецификации (из-за ограничений грамматики), но присутствуют и лишние утверждения (из-за свойств входных данных).

Исследования, описанные в работе [8], показывают, что тестовые наборы разумного размера позволяют получить с помощью *Daikon* достаточно качественные спецификации. С ростом тестового набора результаты вывода утверждений, как правило, улучшаются (однако, размер не является определяющим критерием качества тестового набора в этом случае). Кроме того, авторы исследований утверждают, что системные тесты порождают гораздо более качественные результаты, чем модульные тесты, а случайное тестирование вообще малопригодно для вывода утверждений. Еще один интересный вывод из этого эксперимента: *Daikon*, в основном, выводит простые низкоуровневые утверждения, в то время как заключение свойств программы на высоком уровне абстракции ему недоступно.

Эксперименты с участем промышленных программистов, описанные в работе [8], показали, что число релевантных утверждений, выведенных системой *Daikon*, в среднем больше числа утверждений, записанных разработчиком. Половина опрошенных пользователей посчитали результаты вывода утверждений полезными, более половины согласились, что удалять «лишние» выведенные утверждения достаточно легко.

Кроме того, в работе [8] содержится оценка трудоемкости вывода утверждений в детекторе *Daikon*. Время вывода утверждений растет линейно с увеличением числа наблюдений точек программы в файле истории. Другими сло-

вами, время примерно пропорционально размеру исходной системы и размеру тестового набора (или времени выполнения системы). Поскольку в грамматику *Daikon* входят унарные, бинарные и тернарные утверждения, время вывода утверждений для одной точки программы зависит кубически от числа переменных в этой точке. Кроме того, время вывода пропорционально числу шаблонов утверждений в грамматике.

Вопрос масштабируемости является одним из центральных при обсуждении системы *Daikon*. Основная проблема заключается в том, что размер файла истории выполнения небольшой программной системы может достигать нескольких гигабайт. Первоначально в детекторе *Daikon* применялся многопроходный алгоритм вывода утверждений, при этом вся история загружалась в оперативную память. Использование такого подхода для программных систем среднего и большого размера практически невозможно. Для решения этой проблемы в работе [13] был предложен инкрементальный алгоритм, который позволяет производить вывод утверждений в один проход ценой некоторой потери эффективности. При этом историю не только не требуется загружать в оперативную память, но и хранить на диске, так как все значения переменных могут передаваться детектору утверждений по мере их появления. Отметим однако, что в большинстве случаев вывод утверждений в различных точках программы производится независимо (исключение составляют точки программы, связанные отношением наследования; этот вопрос обсуждается в разд. 2.5). Поэтому в качестве альтернативы использованию менее эффективного инкрементального алгоритма можно предложить обработку не всей системы сразу, а отдельных классов, компонентов классов или точек программы по очереди.

Одно из самых интересных известных применений системы *Daikon* — улучшение качества тестовых наборов. Под этим подразумевается широкий спектр задач: классификация тестовых случаев, выбор тестовых случаев, выявляющих ошибки, автоматическая генерация, пополнение и минимизация тестовых наборов.

В работе [14] описан один из подходов к улучшению качества тестовых наборов при помощи динамического вывода утверждений и описано инструментальное средство *Eclat*, реализующее этот подход. Сначала существующий тестовый набор, не выявляющий ошибок, используется для вывода спецификации программы с помощью системы *Daikon*. Затем выведенная спецификация

используется для автоматической классификации новых тестовых случаев, которые генерируются системой случайного тестирования:

- если при выполнении подпрограммы на новом тестовом случае выведенные предусловие и постусловие были соблюдены, то случай классифицируется как *нормальное выполнение*;
- если было нарушено выведенное предусловие, а постусловие соблюдено, то случай классифицируется как *новый*;
- если предусловие было соблюдено, а постусловие нарушено, считается, что данный случай *выявляет ошибку*;
- наконец, если и предусловие и постусловие были нарушены, случай считается *нелегальным*.

Из всех тестовых случаев, выявляющих ошибки, *Eclat* выбирает по одному на каждое нарушенное постусловие. Анализ этого минимизированного набора тестов позволяет разработчику гораздо эффективнее находить и исправлять ошибки, чем при использовании обычного случайного тестирования. Кроме того, тестовые случаи, которые были классифицированы как *новые*, могут быть добавлены к исходному тестовому набору, так как они предположительно тестируют то поведение программы, которое не было затронуто исходным набором.

В работе [15] была предложена техника *операционной разницы* для генерации, пополнения и минимизации тестовых наборов. Ее основная идея в том, что спецификация, выводимая системой *Daikon*, «стремится» к искомой полной спецификации (насколько она может быть выражена в грамматике *Daikon*) с ростом разнообразия тестовых случаев в наборе. Техника операционной разницы — это своеобразный итерационный метод поиска полной спецификации. Этот процесс может начинаться с некоторого уже существующего тестового набора или с нуля. На каждой итерации случайным образом генерируется один или несколько новых тестовых случаев, после чего производится динамический вывод утверждений. Процесс завершается, когда выведенная спецификация остается неизменной в течении определенного числа итераций. Для минимизации существующего тестового набора методом операционной разницы из набора, напротив, удаляются тестовые случаи до тех пор, пока выводимая спецификация остается неизменной.

Описанные выше методы улучшения качества тестовых наборов не требуют наличия исходной спецификации, созданной разработчиком. С одной стороны, это достоинство, которое позволяет применять их для основной массы существующего ПО. С другой стороны, это приводит к тому, что в полученных такими методами тестовых наборах могут содержаться нелегальные случаи исполнения программы. Авторы работы [14] отмечают, что созданные разработчиком и выведенные автоматически спецификации могли бы органично дополнять друг друга в процессе классификации тестовых случаев. В работе [15] упоминается, что техника операционной разницы может быть усовершенствована с целью использования преимуществ наличия частичной спецификации, созданной разработчиком.

Авторы работ [16, 17] рассматривают зависимость выведенной спецификации от тестового набора с другой стороны. Они задаются вопросом, какими свойствами должен обладать тестовый набор для того, чтобы качество выведенных утверждений было наилучшим. В работе [16] утверждается, что существующие структурные критерии покрытия кода, такие как покрытие условий или покрытие пар определение-использование, неадекватны для рассматриваемой цели. В работе [17] предлагается новый критерий покрытия, разработанный на основе покрытия пар определение-использование. Однако этот критерий не получил широкого распространения, так как его вычисление слишком трудоемко. Кроме того в работе [16] показано, что задачу минимизации числа неверных выведенных утверждений можно свести к стандартной задаче генерации тестовых данных. А именно: если необходимо установить истинность некоторого выведенного утверждения в определенной точке программы, можно вставить в эту точку инструкцию ветвления, условием которой будет служить проверяемое утверждение. После этого можно использовать любой метод генерации тестовых данных для установления возможности или невозможности покрытия ложной ветви инструкции. Если это окажется возможным, значит исходное утверждение было неверным. Отметим, что существующие техники генерации исходных не позволяют решить эту задачу в общем случае.

Исследование еще одного интересного применения динамического вывода утверждений содержится в работе [18]. Ее авторы задались целью проверить, возможно ли верифицировать выведенные спецификации методом статического анализа, а также доказывать на их основе полезные свойства программы, такие

как, например, отсутствие исключений во время выполнения. Авторы пробовали верифицировать код на языке *Java*, аннотированный системой *Daikon*, с помощью статического анализатора *ECS/Java*. Эксперимент показал, что достаточно небольшие тестовые наборы позволяют выводить спецификации, которые могут быть статически доказаны, а также на их основе может быть доказано свойство отсутствия исключений. Авторы полагают, что совместное использование статических и динамических подходов к проверке программ крайне эффективно.

В литературе также обсуждалась проблема взаимодействия динамического вывода утверждений с наследованием и полиморфизмом (в контексте языка *Java*). Один из затронутых вопросов — вывод утверждений о динамических типах сущностей. В стандартную функциональность детектора *Daikon* включена возможность выявления переменных, которые на протяжении всего выполнения в данной точке программы имеют один и тот же динамический тип, отличный от статического. Например, пусть известно, что на выходе из функции `item` класса `STACK`[2] результат, объявленный как `ANY` (в *Java* этот класс называется `Object`) всегда имеет тип `INTEGER`. Эту информацию можно использовать при повторном проходе вывода утверждений, обращаясь с результатом функции так, как будто он объявлен с типом `INTEGER`, например, вывести утверждение `Result >= 0`.

В работе [19] продолжены исследования в этом направлении и предложено инструментальное средство *Turnip* — модификация *Daikon*, которая позволяет выводить утверждения, зависящие от динамического типа сущности. Например, для процедуры `scale (amount: INTEGER)` отложенного класса `FIGURE` может быть выведено следующие постусловие:

```
ensure
  generator.is_equal ("CIRCLE") implies radius = old radius * amount
  generator.is_equal ("RECTANGLE") implies width = old width * amount
  generator.is_equal ("RECTANGLE") implies height = old height * amount
  ...
```

В работе [20] обсуждается другая проблема взаимодействия вывода утверждений и наследования: более актуальная и более согласованная с принципами объектно-ориентированного программирования. Авторы этой работы указыва-

[2] В настоящей работе для описания всех элементов исходного кода программ используются соглашения, принятые в языке *Eiffel*. При необходимости приводятся эквиваленты этих элементов из языка *Java*.

ют на противоречия принципу подстановочности, которые могут возникать при использовании стандартной техники вывода утверждений в контракте наследования подтипов. В этой работе в качестве языка программирования рассматривается *Java*, а в качестве языка спецификации — *JML*. Принцип подстановочности в *JML* реализуется следующим образом.

Пусть в классе A определен компонент f:

```
f is
  require P
  ensure Q
end
```

В этом случае предусловие $pre_{A.f}$ и постусловие $post_{A.f}$ компонента f определяются по формулам:

$$pre_{A.f} = P$$
$$post_{A.f} = P \implies Q$$

Пусть у класса A есть подкласс B, который переопределяет компонент f следующим образом:

```
f is
  require else R
  ensure then S
end
```

Тогда:

$$pre_{B.f} = P \vee R$$
$$post_{B.f} = (P \implies Q) \wedge (R \implies S)$$

Рассмотрим реальный пример из библиотеки *Base* — стандартной библиотеки структур данных языка *Eiffel*. Класс `LINKED_LIST` определяет процедуру `extend (v: like item)`, которая вставляет объект `v` в конец списка. Его наследник, класс `LINKED_STACK`, переопределяет эту процедуру так, что новый элемент вставляется в начало списка, которое соответствует вершине стека. Стандартная техника динамического вывода утверждений в этом случае может породить следующую спецификацию:

```
class LINKED_LIST
  ...
```

```
  extend (v: like item) is
    require
      extendible
    ensure
      last = v
    end
end

class LINKED_STACK
  ...
  extend (v: like item) is
    require else
      extendible
    ensure then
      first = v
    end
end
```

Тогда по правилам *JML* полное постусловие процедуры `extend` в классе `LINKED_STACK` будет

$$(\texttt{old extendible} \implies \texttt{first = v}) \wedge (\texttt{old extendible} \implies \texttt{last = v})$$

или

$$\texttt{old extendible} \implies \texttt{first = v} \wedge \texttt{last = v}$$

Это утверждение не может быть всегда верным по завершении работы процедуры `extend`. Таким образом, использование стандартной техники динамического вывода утверждений привело к противоречию с принципом подстановочности. Такие противоречия вызывают особенно серьезные проблемы, если предполагается обрабатывать выведенные спецификации автоматически, например, верифицировать статическим анализатором.

Для решения этой проблемы авторы работы [20] предлагают технику вывода утверждений, согласованную с принципом подстановочности. Каждому полиморфному вызову во время выполнения программы сопоставляется его *статический получатель* (тот компонент, который известен клиенту вызова) и *динамический получатель* (компонент, который действительно вызывается в результате полиморфизма). Вывод утверждений предлагается производить в два прохода. На первом проходе значения переменных перед каждым поли-

морфным вызовом используются для вывода предусловий динамического получателя вызова и всех компонентов, которые он переопределяет вплоть до статического получателя. На втором проходе значения переменных после каждого полиморфного вызова используются для вывода постусловий динамического получателя и всех компонентов, которые он переопределяет до тех пор, пока данный вызов удовлетворяет их предусловиям (на этом этапе необходимо знать предусловия компонентов, именно поэтому вывод утверждений производится в два прохода). При таком подходе противоречий с принципом подстановочности не возникает, так как при выводе свойств поведения класса-предка учитывается поведение его потомков.

Глава 2.

Вывод утверждений в языке программирования *Eiffel*

В данной главе подробно рассматривается техника динамического вывода утверждений в языке *Eiffel*. Как было упомянуто выше, собственно детектор утверждений в системе *Daikon* универсален и не зависит от языка программирования. Поэтому языковая специфика в основном касается выбора точек программы и интересных переменных, а также оснащения исходной системы инструкциями записи истории выполнения.

Данная глава описывает особенности предлагаемой техники вывода утверждений для языка *Eiffel* в сравнении с наиболее широко используемой и наиболее близкой техникой для языка *Java*, реализованной в интерфейсе *Chicory*. Этот интерфейс разработан авторами детектора *Daikon* и входит в стандартную поставку системы *Daikon*.

2.1. Назначение вывода утверждений

Динамический вывод утверждений показал свою эффективность для языков программирования, в которых нет встроенных механизмов спецификации. Но зачем ставить вопрос о применимости вывода утверждений в *Eiffel*, если в нем уже есть проектирование по контракту? Дело в том, что, как было упомянуто во введении, спецификации разработчиков не являются полными и, как правило, довольно слабы.

При этом качество спецификаций для разных видов утверждений неодинаково: предусловия разработчиков обычно полны в отличие от постусловий и инвариантов классов, в то время как инварианты циклов чаще всего вообще отсутствуют. Более высокое качество предусловий объясняется тем, что слишком слабое предусловие — это ошибка в программе, в то время как слишком слабое постусловие или инвариант не ошибка, а всего лишь частичная спецификация. Инварианты циклов (также, как и проверки) считаются менее инте-

ресными, чем остальные виды утверждений, так как они описывают свойства реализации класса, а не его интерфейса. Эти свойства имеют значение лишь для разработчика данного класса, но не для разработчиков его клиентов и наследников. Другая причина малой популярности инвариантов циклов в том, что нетривиальные циклы, требующие спецификации инварианта, встречаются лишь в небольшой доле программ, реализующих сложные алгоритмы. В основном циклы в программах на *Eiffel* настолько просты, что от указания инварианта понять их станет только труднее.

Кроме неполноты контрактов разработчика есть и другая причина для применения динамического вывода утверждений в *Eiffel*. Спецификации разработчика отражают желаемые, искомые свойства программной системы, в то время как выведенные спецификации отражают ее реальные свойства, а также свойства контекста, в котором система выполняется. Эта особенность открывает для выведенных контрактов такие варианты использования, на которые спецификации разработчика не годились бы, даже если бы они были идеальными.

Итак, перечислим основные предполагаемые варианты использования динамического вывода утверждений в языке *Eiffel*.

Усиление контрактов разработчика. Под усилением контрактов понимается, в основном, усиление постусловий компонентов и инвариантов классов. Усиление (а в большинстве случаев, генерация с нуля) инвариантов циклов также относится к этому варианту использования, однако, по перечисленным выше причинам, это менее интересно. В общем случае усиленная выведенными утверждениями спецификация все еще не будет полной, однако в контексте динамических методов проверки корректности (мониторинга утверждений во время выполнения, тестирования, где контракты выступают в качестве оракулов) ее качество улучшится. Негативная сторона усиления контрактов разработчика при помощи динамического вывода утверждений в том, что даже использование идеального инструмента и тестового набора может приводить к *чрезмерной спецификации*, когда в усиленном контракте фигурируют не только свойства интерфейса класса, но и свойства его реализации. Поскольку никаких формальных критериев чрезмерной спецификации не существует, от нее невозможно избавиться автоматически.

Исправление контрактов разработчика. Здесь речь идет, в основном, об усилении предусловий. Как было упомянуто выше, слишком слабое предусловие компонента — это ошибка, но бывает, что эта ошибка не выявляется при тестировании, так как компонент секретный (не экспортирован ни одному клиенту) и вызывается всегда из того же класса, где он определен, и всегда из легального контекста. Такая ошибка может дать о себе знать, когда у класса появятся наследники, которые смогут вызывать секретный компонент из другого контекста. Динамический вывод утверждений помогает находить и исправлять такие ошибки заблаговременно.

Улучшение качества тестовых наборов. В языке *Eiffel* могут использоваться те же техники генерации, пополнения и минимизации тестовых наборов, которые применяются для других языков, например, описанные в работах [14, 15]. Более того, эти техники могут быть модифицированы так, чтобы получить наибольшую выгоду от наличия частичных спецификаций разработчика. Наиболее эффективно могут быть использованы предусловия разработчика, которые почти всегда полны. Это позволит отфильтровать нелегальные случаи работы программы, преодолев тем самым главный недостаток существующих методов. Кроме того, можно предложить новые подходы к улучшению качества тестовых наборов, которые с самого начала основывались бы на наличии частичным спецификаций.

Помощь при статическом анализе. Этот вариант использования, вообще говоря, не специфичен для *Eiffel*. Предполагается более вероятным, что контракты разработчика, усиленные выведенными утверждениями, смогут быть верифицированы статическим анализатором. Эти предположения целиком основываются на результатах работы [18].

Все перечисленные выше варианты использования служат общей цели поиска и исправления программных ошибок и, в конечном счете, обеспечения корректности ПО.

Проведенный автором эксперимент по применению динамического вывода утверждений к программам на языке *Eiffel*, в основном, посвящен исследованию двух первых вариантов использования. Что касается последнего варианта, практических исследований в этой области пока не проводилось.

2.1.1. Метод сведения предусловий

Вернемся к улучшению качества тестовых наборов. Для этой цели в настоящей работе предлагается метод *сведе́ния предусловий*. В соответствии с этим методом тестовые случаи следует добавлять в набор до тех пор, пока выведенное предусловие каждого тестируемого компонента не станет слабее предусловия разработчика.

Метод может использоваться как в ручном, так и в автоматизированном варианте. В первом случае на каждой итерации разработчик сравнивает исходное предусловие с выведенным и специально подбирает новые тестовые случаи так, чтобы нарушить выведенное предусловие, не нарушая исходного. В автоматизированном варианте новые тестовые случаи генерируются случайным образом, при чем в тестовый набор попадают только те из них, которые нарушают выведенное предусловие, но не нарушают исходное. Процесс останавливается тогда, когда на протяжении определенного числа итераций ни один новый тестовый случай не попадает в набор. По предположению автора, основная проблема этого алгоритма состоит в том, что, если область значений, удовлетворяющих предусловию разработчика и не удовлетворяющих выведенному предусловию, мала, сгенерировать значения из этой области случайным образом проблематично. Предположительно, эффективность метода можно повысить при помощи какой-либо техники генерации контр-примеров. На данный момент автоматизированный вариант метода сведения предусловий не реализован, ручной вариант был апробирован только на искусственных примерах (ошибки в программу вносились специально). Этот эксперимент рассмотрен в разд. 4.4.

Метод сведения предусловий позволяет получить тестовый набор с высоким разнообразием входных данных. Умышленное нарушение выведенного предусловия часто позволяет протестировать важный специальный случай, воспроизведение которого при случайном тестировании маловероятна. Отчасти это объясняется тем, что грамматика утверждений и правила их вывода в *Daikon* составлялись с учетом природы спецификаций, которые обычно пишут разработчики. Например, если случайно сгенерировано сто значений некоторой целочисленной переменной по равномерному распределению в диапазоне $[-100; 100]$ и среди этих значений не было нуля, детектор *Daikon* сообщит этом. В то же время, если среди них не было, к примеру, числа -73, *Daikon* с большой вероят-

ностью об этом умолчит. Создатели детектора знали, что значение ноль часто является важным специальным случаем, поэтому оно обрабатывается особенным образом.

Процесс улучшения качества тестового набора методом сведения предусловий не мешает, а напротив, отлично сочетается с процессом усиления контрактов разработчика. Поскольку метод сведения предусловий предполагает намеренное опровержение неверных выведенных предусловий (тех, которые являются свойствами тестового набора, а не исходной программной системы), контракты, выведенные на результирующем тестовом наборе будут более высокого качества, чем для исходного тестового набора.

2.2. Точки программы и переменные

Цель динамического вывода утверждений — выявление свойств состояния программы в определенных ее точках. Рассмотрим подробнее понятия *точки программы* и ее *состояния*.

Для начала представим себе простейший императивный язык программирования, в котором нет ни классов, ни даже подпрограмм, а программа представляет собой последовательность простых инструкций. В таком языке определение указанных выше понятий не составит труда: точка программы — это место в программе между любыми двумя инструкциями, а состояние программы — это множество значений всех ее переменных. При осуществлении динамического вывода утверждений в таком языке, разработчик, скорее всего, был бы заинтересован в свойствах программы не в любой ее точке, а выбрал бы какое-то подмножество точек программы, которые были бы ему интересны. Например: перед первой и после последней инструкции или после каждой инструкции условного прыжка. В эти интересные точки программы следовало бы вставить инструкции (или последовательности инструкций), записывающие в файл истории значения всех переменных программы. На этом процесс оснащения программы был бы закончен, не вызвав никаких проблем.

Однако в языке с подпрограммами, а тем более в объектно-ориентированном языке, все не так просто. Например, уже упоминалось, что интересными точками в частности считаются точки входа в программу. Но это уже не «место в программе между двумя инструкциями». Вход в подпрограмму имеют двоя-

кую природу. Одна его часть лежит на вызываемой стороне, и это место в исходном коде указать легко: перед первой инструкцией подпрограммы. Но другая его часть относится к вызывающей стороне, а мест, где вызывается та или иная подпрограмма, в коде программы может быть сколько угодно. В то же время, для целей вывода утверждений может быть нужна информация, как с вызываемой, так и с вызывающей стороны. Такая ситуация как раз возникает в *Eiffel*, где особую роль играет, является ли целью вызова текущий объект `Current` (в этом случае вызов называется *неквалифицированным*, а в противном случае — *квалифицированным*). Информация такого рода недоступна вызываемой стороне. Поэтому в объектно-ориентированной программной системе некорректно отождествлять точки программы в том смысле, как они понимаются при динамическом вывод утверждений, с определенными местами в исходном коде.

Подчеркнем, что этот вывод не относится к рассмотрению программ в динамике. Хотя объектно-ориентированная программная система по своей природе очень далека от последовательности инструкций и представляет собой множество равноправных модулей, ее *выполнение*, как и любой другой программы, можно представить в виде последовательности чередующихся состояний S_k и инструкций i_k:

$$\{S_0 i_1 S_1 \dots i_k S_k \dots i_n S_n\}$$

При этом любая интересная точка программы будет соответствовать некотрому множеству состояний в этой последовательности или, иначе говоря, мест между *выполнением* двух инструкций. При этом в динамике вся информация, необходимая для вывода утверждений, уже присутствует. Например, если некоторое состояние S_k из приведенной выше последовательности соответствует точке входа в подпрограмму, уже известно, из какого контекста вызывается эта подпрограмма в конкретный момент времени k.

В языке *Eiffel* будем отождествлять точки программы не с местами в исходном коде, а с *утверждениями языка* — инвариантами классов, предусловиями и постусловиями компонентов классов, инвариантами циклов. Это логичное решение, поскольку выведенные утверждение предполагается использовать как формальный текст на *Eiffel* и внедрять в исходный код классов. Поэтому требуется выводить те виды утверждений, которые поддерживаются языком. Для дальнейшего обсуждения удобно дать различным типам точек программы собственные названия (табл. 1).

Таблица 1. Соответствие видов утверждений *Eiffel* типам точек программы

Утверждение	**Точка программы**
Предусловие	Вход
Постусловие	Выход
Инвариант класса	Стабильное состояние
Инвариант цикла	Цикл

Отметим, что вариантам циклов не сопоставляются точки программы, так как они не являются утверждениями, и детектор *Daikon* не может их выводить. В принципе возможно модифицировать детектор и реализовать функцию вывода вариантов, однако на данный момент такая функция отсутствует. Кроме того, точки программы не сопоставляются проверкам. Проверка может находиться между любыми двумя инструкциями в теле подпрограммы, однако очевидно, что не каждая такая точка интересна разработчику. С другой стороны, если проверка уже присутствует в некоторой точке, эта точка становится интересной, но от усиления проверок, созданных разработчиком, все равно нет практической пользы.

В каких условиях в ходе выполнения программной системы наблюдаются те или иные точки программы? Что касается точек «вход» и «выход», ответ ясен (если не брать в расчет полиморфизм; этот вопрос обсуждается в разд. 2.5). «Стабильное состояние» каждого класса наблюдается по правилам *Eiffel* на входе и выходе из всех квалифицированных вызовов компонентов этого класса. Именно в этом случае для определения точки программы необходима информация с вызывающей стороны. Точка «цикл» — своя для каждого цикла — наблюдается после каждого выполнения составных инструкций `from` и `loop` данного цикла.

Расходится ли такой выбор интересных точек программы с решениями, реализованными в других интерфейсах к *Daikon*? В языке *Java* изначально отсутствуют утверждения, поэтому выбор интересных точек программы более произволен. Однако, в соответствии с традицией распространенных языков и методов спецификации, в качестве целевых утверждений для динамического вывода в *Java* также выбраны предусловия, постусловия и инварианты классов.

Инварианты циклов в интерфейсе *Chicory* не поддерживаются. Кроме того, поскольку в *Java* нет формальных правил на этот счет, считается, что стабильное состояние наблюдается на каждом входе и выходе из публичных подпрограмм класса (вне зависимости от квалифицированности вызова), что значительно упрощает реализацию.

Перейдем к обсуждению состояния системы. Поскольку объектно-ориентированное программирование строится на принципе локальности данных, определение состояния программной системы как множества значений всех ее переменных здесь не работает. Для того, чтобы выведенное утверждение было корректным кодом на *Eiffel*, в каждой точке программы в качестве переменных следует использовать только те сущности, к которым правила языка разрешают обращаться из соответствующего утверждения. В инварианте класса можно обращаться только к текущему объекту (`Current`). В утверждениях, определенных для компонента класса, сюда добавляются формальные аргументы этого компонента, а в случае постусловия функции — еще и ее результат (`Result`). В инвариантах циклов, кроме того, можно обращаться к локальным переменным объемлющей подпрограммы. Особый случай представляет собой предусловие процедуры создания класса (в других языках она называется конструктором). Это единственное утверждение, в котором запрещено обращаться к текущему объекту, так как в процессе создания объекта выполнить произвольные требования к его состоянию все равно невозможно. В табл. 2 приведен полный список переменных, допустимых для использования (или иначе *видимых*) в утверждениях каждого вида.

2.3. Развертывание переменных

Обсуждение в предыдущем разделе пришло к тому, что в файл истории необходимо записывать значения всех видимых переменных в каждой интересной точке программы. Этого было бы достаточно, если бы все переменные были *примитивных типов* (так в гибридных языках программирования называют типы, не порождаемые классами: логические значения, числа, символы, строки). Однако в языке *Eiffel* вообще нет примитивных типов: все типы основаны на классах, и значение любой переменной является объектом. Как вывести в файл истории объект так, чтобы детектор *Daikon* смог его прочитать?

Таблица 2. Видимые переменные

Утверждение	**Переменные**
Инвариант класса	`Current`
Предусловие	`Current`, формальные аргументы
Предусловие процедуры создания	Формальные аргументы
Постусловие процедуры	`Current`, формальные аргументы
Постусловие функции	`Current`, формальные аргументы, `Result`
Инвариант цикла в теле процедуры	`Current`, формальные аргументы, локальные переменные
Инвариант цикла в теле функции	`Current`, формальные аргументы, локальные переменные, `Result`

Детектор *Daikon* способен выводить утверждения о переменных, принадлежащих к заранее определенному небольшому набору *печатаемых типов* (название обусловлено тем фактом, что значения переменных этих типов без труда можно напечатать в файле истории). Этот набор составлен, в основном, по аналогии с множеством примитивных типов языка *Java*, но с некоторыми добавления. К печатаемым типам *Daikon* относятся:

- `boolean` — логический тип;
- `int` — целочисленный тип;
- `double` — тип чисел с плавающей точкой;
- `String` — строковый тип (пишется с большой буквы по аналогии с классом `String` в *Java*);
- `hashcode` — специальный тип, предназначенный для идентификаторов объектов (например, их адресов);
- T`[]` (где T — любой печатаемый тип) — массив.

Для того, чтобы записать значение переменной в понятном детектору *Daikon* виде, необходимо представить ее в виде множества печатаемых переменных. Эта операция и называется *развертыванием*.

Какой набор печатаемых переменных требуется, чтобы описать все интересные свойства произвольной переменной? Известно, что объект полностью определяется множеством его полей (значений атрибутов класса). Именно этот простой принцип лежит в основе реализации развертывания в *Java*-интерфейсе *Chicory*. Все типы в языке *Java* делятся на примитивные и ссылочные. Значения переменных примитивных типов, а также строки, записываются в файл истории без развертывания, поскольку они соответствуют печатаемым типам *Daikon*. Каждой переменной ссылочного типа *Chicory* сопоставляет ее адрес (в *Java* для получения адреса предназначен запрос `hashcode` — отсюда и название соответствующего печатаемого типа) и вызовы всех атрибутов ее базового класса. Так как атрибуты сами могут иметь ссылочный тип, операция развертывания в свою очередь применяется и к ним. Этот процесс мог бы продолжаться до бесконечности, но на практике его ограничивают некоторым фиксированным числом итераций (обычно одной или двумя). Это число называется *глубиной* развертывания. Опыт показывает, что большая глубина развертывания приводит не только к значительному увеличению времени работы детектора, но и к снижению качества выведенной спецификации. Причина в том, что утверждения, включающие такие переменные, как например `a.b.c.d`, очень редко бывают релевантными. Адрес переменной здесь служит ее идентификатором: он используется исключительно для установления равенства переменных (ссылочные переменные считаются равными, если они ссылаются на один и тот же объект), никакие другие свойства адреса детектор *Daikon* не выводит.

Специальная техника развертывания применяется для контейнеров. Состояние контейнера выводится в файл истории в виде последовательности хранящихся в нем значений, например `[1, 2, 3]` или `["hello", "good bye"]`. Именно для этой цели в *Daikon* введен печатаемый тип массив. Такое представление позволяет детектору проверять целый ряд свойств контейнеров, например, упорядоченность, присутствует ли в контейнере заданный элемент или является ли одна последовательность значений подпоследовательностью другой. Представление в виде массива целесообразно использовать для таких кон-

тейнеров как вектор или список, которые предоставляют доступ ко всем своим элементам.

На рис. 2 приведен пример развертывания переменной `Current` в классе `LINKED_STACK` по правилам *Chicory*. На рисунке справа от имени каждой переменной указано ее значения (для примитивных переменных) или адрес (для ссылочных). Стрелки ведут от объекта к его полям.

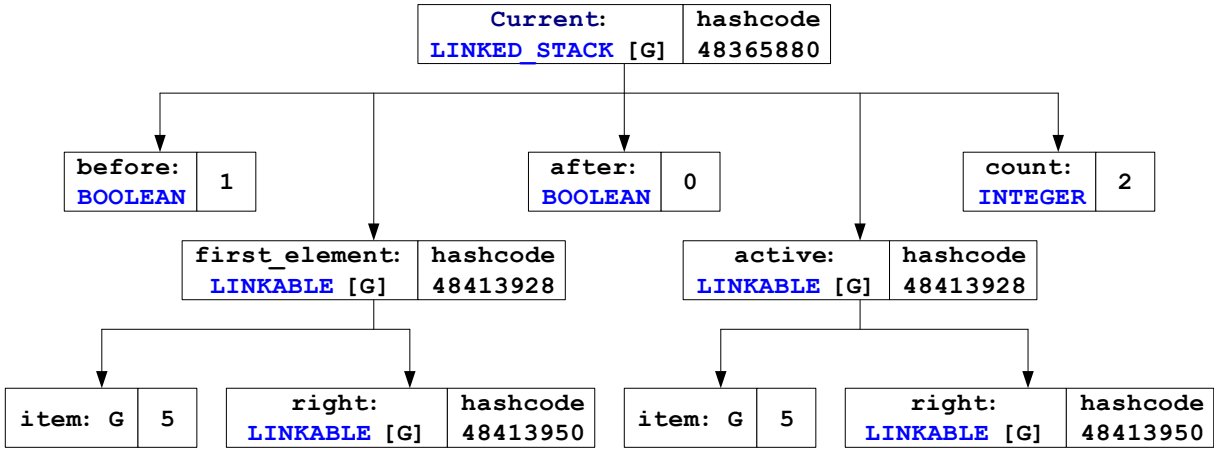

Рис. 2. Пример развертывания по правилам *Chicory* (с глубиной два)

В языке *Eiffel* в точности такой подход к развертыванию переменных является недопустимым по нескольким причинам.

Правила экспорта. Выше упоминалось, что код программы, аннотированный выведенными утверждениями, должен оставаться корректным текстом на *Eiffel*. Правила языка запрещают использовать в утверждении вызовы тех компонент, которые недоступны объемлющему классу утверждения или какому-либо его потенциальному гаранту (под *гарантом* утверждения понимается класс, который должен обеспечить его истинность). Так, поскольку за истинность предусловия подпрограммы отвечает клиент, то в ее предусловии могут использоваться вызовы только тех компонентов, которые доступны всем потенциальным клиентам этой подпрограммы, иначе говоря, всем классам, которым подпрограмма экспортирована. В противном случае от клиента требовалось бы обеспечить то, что он не может даже проверить. При использовании подхода, реализованного в интерфейсе *Chicory*, выведенные предусловия публичных подпрограмм могут содержать обращения к секретным атрибутам. Такие утверждения являются некорректными, как формально, так и концептуально.

Принцип унифицированного доступа. В языке *Eiffel* существует принцип *унифицированного доступа*, в соответствии с которым вызов атрибута неотличим для клиента от вызова функции без аргументов. Этот принцип обеспечивает свободу выбора реализации запроса без необходимости изменения интерфейса, и, следовательно, способствует инкапсуляции. В подходе *Chicory* при развертывании используются атрибуты, но не используются функции без аргументов, сто концептуально противоречит принципу унифицированного доступа.

Развернутые типы. Все классы *Eiffel* делятся на ссылочные и *развернутые* (этот термин не имеет ничего общего с развертыванием переменных). Объявляя класс развернутым, разработчик наделяет его экземпляры семантикой значений вместо семантики ссылок. В частности, экземпляры развернутых типов сравниваются по значению, а не по адресу. Поэтому, в отличие от объектов языка *Java*, адрес уже не может служить универсальным идентификатором объекта в *Eiffel*.

Для разрешения указанных противоречий в настоящей работе предлагается новая модель развертывания переменных, согласованная с правилами и философией языка *Eiffel*. *Полную развернутую форму* $U(x)$ переменной x формально определим как множество печатаемых переменных, состоящее из ее *идентификатора* $I(x)$ и объединения развернутых форм ее *дочерних* переменных $C(x)$:

$$U(x) = I(x) \cup \bigcup_{y \in C(x)} U(y)$$

Полная развернутая форма в общем случае содержит бесконечное число печатаемых переменных. Как было упомянуто выше, процесс развертывания разумно ограничить фиксированным числом итераций и перейти к $U_n(x)$ — *развернутой форме глубины* n, которую можно формально определить следующим образом:

$$U_0(x) = I(x)$$

$$U_k(x) = U_{k-1}(x) \cup \bigcup_{y \in U_{k-1}(x)} \left(\bigcup_{z \in C(I^{-1}(y))} I(z) \right)$$

2.3.1. Идентификатор переменной

Идентификатор переменной, как и раньше, служит исключительно для проверки на равенство. Однако теперь идентификатор определяется по-разному в зависимости от типа переменной. Идентификатор переменной ссылочного типа — это ее адрес. Переменные развернутых типов считаются равными в *Eiffel*, если и только если все поля присоединенных к ним объектов попарно равны. Поэтому в качестве идентификатора экземпляра развернутого типа предлагается выбрать некоторую функцию свертки идентификаторов значений всех его полей:

$$I(x) = F(\text{field } (x, 1), \ldots, \text{field } (x, n))$$

Здесь функция field (x, i) возвращает i-е поле объекта x, состоящего из n полей. Поскольку разрядность печатаемого типа `hashcode` в *Daikon* фиксирована, а число и типы полей объекта произвольны, при вычислении идентификатора переменной развернутого типа с неизбежностью происходит потеря информации. Это может привести к коллизиям, когда объекты, на самом деле не являющиеся равными, будут иметь одинаковые идентификаторы.

Однако эта проблема не так серьезна, как кажется на первый взгляд. Во-первых, для того, чтобы детектор в некоторой точке программы вывел утверждение о равенстве двух переменных, их значения должны быть равны при каждом наблюдении этой точки в истории. Если k раз при наблюдении данной точки программы значения переменных в действительности не были равны, а вероятность коллизии идентификаторов для одной пары объектов равна p, то вероятность вывода неверного утверждения равенства есть p^k.

Во-вторых, учитывая наблюдение, что значения переменных во время исполнения программных систем распределены по их допустимым диапазонам неравномерно, можно подобрать функцию свертки так, чтобы уменьшить вероятность коллизий. Значения различных переменных в одной точке программы обычно близки друг к другу: вещественные числа часто бывают одного порядка, адреса объектов принадлежат к одной области адресного пространства, целые числа вообще как правило невелики по модулю, причем отрицательные целые значения встречаются значительно реже. Приняв гипотезу, что битовые представления переменных в одной точке программы близки друг к другу, можно использовать битовый сдвиг различных полей объекта на разное число бит, что-

бы разнести их «сферы влияния» в идентификаторе. На практике была апробирована функция свертки следующего вида:

$$F(x) = \bigoplus_{i=1}^{n} \text{field } (x, i) \ll \left[\frac{c}{n}\right] \times (i - 1)$$

Здесь c — разрядность печатаемого типа `hashcode`, $\oplus$ — сложение по модулю два (побитовое «или»), $\ll$ — операция циклического битового сдвига влево.

В целях оценки эффективности этой функции автором был проведен небольшой эксперимент. Случайным образом было сгенерировано сто тысяч объектов с целочисленными полями. Значения полей объектов выбирались в соответствии с распределением, эквивалентным равномерному распределению $U[-50; 50]$, с точностью до того, что неотрицательные значения встречались вдвое чаще отрицательных. Сто тысяч объектов порождает $100\,000 \cdot 99\,999/2 \approx 5 \cdot 10^9$ пар объектов. Для объектов с двумя полями коллизий не возникло, для объектов с пятью полями была обнаружена одна коллизия. Этот результат дает грубую оценку вероятности коллизии $p \approx 2 \cdot 10^{-10}$, что очень неплохо.

Кроме естественного деления классов в *Eiffel* на ссылочные и развернутые оказалось удобно выделить в отдельную группу классы, соответствующие печатаемым типам *Daikon*: `BOOLEAN`, `INTEGER`, `NATURAL`, `REAL`, `CHARACTER`, `STRING`. Объекты всех этих классов имеют строковое представление, которое их полностью характеризует и может быть выведено в файл истории. Поэтому в качестве идентификатора переменной в этом случае может быть использовано ее значение в привычном смысле. Отметим, что класс `STRING` выбивается из приведенного выше ряда, так как является ссылочным. Действительно, строки в *Eiffel* сравниваются по адресу, поэтому было бы неправильно считать идентификатором строковой переменной ее значение. Принято решение различать переменные «ссылка на строку» и «значение строки». Первая является обыкновенной ссылочной переменной, а вторая — печатаемой, и выступает по отношению к первой в роли дочерней переменной специального вида. Таким образом, равенства строковых переменных выводятся на основании ссылок, а все остальные утверждения, специфичные для строк — на основании их значений, что и требовалось. Если детектор выводит равенство значений строк, оно преобразуется в вызов функции `is_equal`, сравнивающей объекты по значению.

Чтобы подытожить изложенное выше, приведем таблицу идентификаторов переменных в зависимости от их типа (табл. 3). Идентификаторы записаны в виде выражений на языке *Eiffel*.

Таблица 3. Идентификаторы различных типов переменных

Класс-генератор переменной `x`	**Идентификатор**
Печатаемый	`x`
Непечатаемый ссылочный	`$x`
Непечатаемый развернутый	`F (field (x, 1), ..., field (x, n))`

Важно отметить, что сравнения в *Eiffel* полиморфны: будет ли сравнение производиться по адресу или по значению, зависит от динамических типов объектов. Поэтому вид идентификатора — это динамическое свойство и относится оно к объекту, а не к переменной. Переменная, статический тип которой является ссылочным, может в процессе выполнения присоединяться к объектам развернутых и печатаемых типов. Поэтому правило, по которому вычисляется идентификатор, также должно определяться во время выполнения.

2.3.2. Дочерние переменные

Дочерние переменные должны отражать те свойства родительской переменной, которые доступны и интересны объемлющему классу точки программы и гаранту соответствующего утверждения. Множество дочерних переменных строится из надлежащим образом экспортированных запросов (атрибутов и функций) базового класса родительской переменной. Иначе говоря, в постусловиях, инвариантах классов и циклов используются запросы, экспортированные объемлющему классу, а в предусловиях — еще и всем потенциальным клиентам объемлющего компонента.

У переменных печатаемых типов по соглашению не дочерних переменных. Развертывание печатаемых переменных не противоречит и модели, просто считается, что всю необходимую информацию об этих переменных детектор утверждений может получить из их идентификаторов.

Множество дочерних переменных, в отличие от идентификатора — свойство статическое, определяемое базовым классом переменной. Это очевидно,

если вспомнить, что множество запросов, которые допустимо вызывать на той или иной переменной, определяется во время компиляции.

На рисунке рис. 3 приведен пример развертывания переменной `Current` на входе в один из экспортированных компонентов класса `LINKED_STACK` по правилам, прелагаемым в настоящей работе. Справа от имени каждой переменной указано значение ее идентификатора. Стрелки ведут от родительских к дочерним переменным. Сравнив рис. 2 и рис. 3, можно заключить, что на последнем отражены свойства стека, более релевантные для его клиентов.

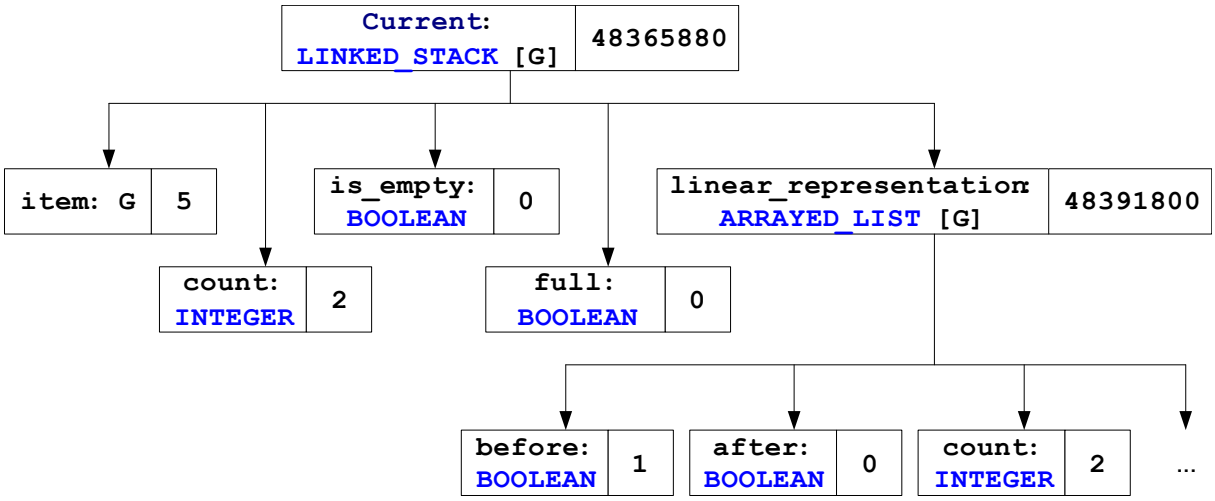

Рис. 3. Пример развертывания в предлагаемой модели (с глубиной два)

В соответствие с принципом унифицированного доступа вполне естественно использование для порождения дочерних переменных функций без аргументов. Предложенная модель допускает также использование функций с аргументами. В качестве фактических аргументов могут выступать другие переменные из той же точки программы, а также небольшой набор констант и простых выражений. По предположению автора, использование функций с аргументами при развертывании может серьезно улучшить полноту выводимых спецификаций, однако значительный рост числа переменных в каждой точке программы может свести на нет это преимущество. Если изначально в некоторой точке программы имеется n переменных и к ним добавить m функций с одним аргументом (для простоты будем считать, что все переменные и аргументы функций одного типа), причем при конструировании дочерней переменной разрешается использовать функцию с аргументом только один раз, то в результате получит-

ся $n + mn$ переменных. Если же каждая из функций имеет p аргументов, то результирующее число переменных равно $n + mn^p$.

Пока что возможность использования функций с аргументами проверялась лишь на одном небольшом классе, причем оснащение производилось в ручную (разд. 4.3).

Использование функций при развертывании порождает сразу несколько проблем. Во-первых, инструкции записи значений переменных в файл истории не должны изменять ничего, кроме этого файла. В противном получится ситуация из области квантовой физики, когда процесс измерения влияет на его результат. Поэтому, если инструкции вывода содержат вызовы функций, то эти функции должны быть *чистыми* (не иметь побочных эффектов). К счастью, в языке *Eiffel* существует принцип *разделения команд и запросов*, в соответствии с которым функции не должны иметь абстрактных побочных эффектов [1, глава 23]. Хотя этот принцип не навязывается языков (поскольку не существует формальных признаков абстрактного побочного эффекта), большинства функций в существующем коде на *Eiffel* действительно являются чистыми. Для тех редких случаев, когда это не так, можно предложить различные решения, например, явное указание функций, обладающих побочным эффектом и исключение их из процесса развертывания.

Другая проблема состоит в том, что значение функции не всегда может быть вычислено, так как функции в *Eiffel* в общем случае частичны. Оговоримся, что при оснащении системы будем считать предусловия функций, созданные разработчиком, верными. Таким образом вычисление значения функции при развертывании будет считаться возможным тогда и только тогда, когда предусловие разработчика выполнено. В детекторе *Daikon* предусмотрено специальное ключевое слово `nonsensical` («бессмысленный») для тех случаев, когда значение переменной не может быть вычислено. Первоначально эта возможность предназначалась для случаев вызова атрибута на пустой ссылке, иными словами, вызова `a.b` при `a = Void` (в *Java* пустая ссылка называется `Null`). Отметим, что такой вариант использования ключевого слова `nonsensical` также актуален для *Eiffel*. Кроме того, будем обозначать с его помощью значение функции, предусловие которой нарушено.

Важно отметить, что предусловие, проверяемое в процессе развертывания, так же, как и множество дочерних переменных и в отличие от идентификато-

ра, является статическим свойством, зависящим от базового класса переменной. Известно, что классы-потомки при переопределении компонентов предка могут ослаблять предусловия. Однако в том случае, если при полиморфном вызове функции в процессе развертывания ее динамическое предусловие выполняется, а статическое не выполняется, результатом должно быть значение `nonsensical`, так как с точки зрения клиента, которому известно лишь статическое предусловие, вызов не является возможным.

В заключение отметим, что интерфейс *Chicory* предоставляет дополнительную возможность использования функций без аргументов при развертывании. Однако в языке *Java* обсуждаемые выше проблемы решить гораздо сложнее. Во-первых, языки семейства *C* поощряют побочные эффекты в функциях. Поэтому *Chicory* требует от пользователя предоставлять списки чистых функций, что не вполне удобно. Во-вторых, в коде *Java* изначально нет предусловий, поэтому невозможно определить, когда вызов функции легален, а когда нет. Это делает оснащенный код крайне подверженным различного рода ошибкам и возникновению исключительных ситуаций, и, в конце концов, ведет к некорректным результатам вывода утверждений.

2.4. Грамматика утверждений

В главе 1 было упомянуто, что одним из определяющих факторов при динамическом выводе утверждений является грамматика утверждений — набор шаблонов, в которые детектор подставляет подходящие переменные. Если выведенные утверждения предназначены для использования в контексте сложного языка спецификации, богатого разнообразными синтаксическими конструкциями, то и грамматика детектора соответственно должна быть сложной.

Зададимся вопросом, какой должна быть грамматика утверждений для языка *Eiffel*. Требуется, чтобы выведенные спецификации были корректными контрактами языка, то есть состояли из предложений, каждое из которых являлось бы правильным булевым выражением на *Eiffel*. Конечно, возможно выводить утверждения произвольного вида и внедрять их в код в качестве комментариев. Однако от предложений-комментариев пользы гораздо меньше, чем от полноценного формального текста.

Грамматика выражений *Eiffel* довольно проста и включает всего восемь их разновидностей:

- текущий объект (`Current`);
- формальный аргумент;
- локальная переменная;
- результат функции (`Result`);
- равенство (и его отрицание — неравенство);
- вызов запроса (может быть квалифицированным, неквалифицированным или статическим, а также может быть записан в инфиксной или префиксной форме);
- создание объекта (с помощью порождающего вызова, литерала или агента);
- пустая ссылка (`Void`).

Такая простота обеспечивается отсутствием примитивных типов и предопределенных операций над ними. Все операции над числами, символами и строками представляют собой обыкновенные вызовы компонентов. Единственная «привилегия» этих типов — возможность создавать экземпляры при помощи литералов.

Можно представить себе идеальную ситуацию, в которой для формирования информации о точке программы из всех видимых в ней переменных в соответствии с приведенными выше правилами конструируются всевозможные булевы выражения до определенной степени сложности. При этом не было бы необходимости в печатаемых типах (кроме логического), а также в какой бы то ни было грамматике утверждений: детектору осталось бы только проверять переданные ему булевы выражения на истинность. Абсолютно все предположительно интересные свойства переменных определялись бы кодом исходной системы, а не правилами, предопределенными в детекторе. Такая модель теоретически красива, но практически нереализуема: хотя бы потому, что различных литералов для создания чисел и строк необозримо много, а значит почти в каждой точке программы было бы необозримо много переменных.

На практике, конечно, есть необходимость в печатаемых типах и шаблонах утверждений, встроенных в детектор. Однако, хотя идеала невозможно достичь, к нему вполне можно приблизиться. Многие элементы грамматики детектора *Daikon* в контексте вывода утверждений на *Eiffel* оказываются лишними. Один из примеров — *производные переменные*. Этот механизм *Daikon* аналогичен описанным выше дочерним переменным, за тем исключением, что значение производной переменной не фигурирует в файле истории, а вычисляется детектором уже на этапе вывода утверждений. Например, *Daikon* может вычислять такие производные переменные как `s.length` (длина строки) или `a[i]` (элемент массива), где `s`, `a`, `i` — оригинальные переменные подходящих типов.

Детектор располагает информацией лишь о *типе представления* переменной — печатаемом типе, которым представлено значение переменной в файле истории. Ему ничего не известно о настоящем базовом классе переменной и даже о языке программирования, на котором описан этот класс. Поэтому утверждения с участием производных переменных могут содержать вызовы несуществующих запросов и следовательно быть некорректными. С другой стороны, дочерние переменные, создаваемые оснастителем, всегда корректны и с бóльшей вероятностью породят релевантные утверждения.

Другой пример бесполезного элемента грамматики *Daikon* — шаблоны утверждений, включающие битовые операции над целыми числами. Эти операции крайне редко фигурируют в коде на *Eiffel*, и их использование при выводе порождает множество нерелевантных утверждений.

Еще одна сомнительная в контексте *Eiffel* возможность — представление контейнеров в виде последовательностей хранящихся в них значений. Более половины всех шаблонов утверждений в грамматике *Daikon* описывают свойства последовательностей. Однако, поскольку эти свойства одинаковы для всех контейнеров, многие из выведенных утверждений опять же окажутся некорректными. Отказавшись от использования последовательностей, можно значительно сократить время вывода утверждений, поскольку проверка свойств последовательностей является наиболее трудоемкой.

По мнению автора, при выводе спецификаций программных систем на языке *Eiffel* целесообразно отказаться от сложной грамматики утверждений, предопределенной в детекторе, в пользу более интенсивного использования свойств

переменных, заложенных в самой системе. В частности участие функций с аргументами в процессе развертывания может не только улучшить релевантность выведенных спецификаций, но и способствовать повышению их уровня абстракции. Такой подход представляется автору наиболее перспективным решением проблемы, упомянутой в работе [8]: неспособности *Daikon* выводить высокоуровневые, абстрактные утверждения. Это предположение на данный момент не было должным образом проверено на практике, были сделаны лишь первые шаги, описанные в разд. 4.3.

Интенсивное использование функций в развертывании переменных соответствует философии контрактов *Eiffel*, основная выразительная сила которых сосредоточена в способности включать вызовы произвольных функций. Проблему при этом представляет большое число переменных в каждой точке программы (разд. 2.3), которое растет экспоненциально при подключении функций со все большим числом аргументов. Хорошая новость состоит в том, что благодаря прицину разделения операндов и опций подпрограммы *Eiffel*, как правило, содержат очень небольшое число аргументов. По данным, приведенным в [1, глава 23], среднее число аргументов подпрограмм в стандартных библиотеках *Eiffel* находится в пределах от 0,4 для библиотеки *Base* до 0,7 для графической библиотеки *Vision*. Это означает, что включив в процесс развертывания только функции с не более чем одним аргументом, можно достичь очень высокого уровня полноты выводимых контрактов, не сталкиваясь с проблемой комбинаторного взрыва числа переменных.

2.5. Вывод утверждений и наследование

В *Eiffel*, где механизм наследования играет важнейшую роль в конструировании ПО, невозможно обойти вопрос взаимодействия этого механизма с динамическим выводом утверждений.

Один из аспектов этого вопроса, как было упомянуто в главе 1 — вывод информации о динамических типах полиморфных сущностей. Такая информация может быть полезна для отладки или для вычисления динамических метрик системы, но в контрактах объектно-ориентированных компонентов она фигурировать не должна. Упоминание динамических типов в спецификации противоречит самой идее полиморфизма подтипов — единообразной работы с

объектами различных классов через интерфейс их общего предка без необходимости в какой-либо информации о динамических типах.

Вывод динамических типов, реализованный в *Daikon*, мог быть полезен в старых версиях языка *Java*, в которых не было обобщенности, и вследствие этого полиморфизм часто использовался не по назначению. Например, функции получения элементов контейнеров имели статический тип `Object`. Если разработчику требовался контейнер для хранения, к примеру, только целых чисел, каждый раз при получении элемента из него приходилось делать явное преобразование типов (или в терминах *Eiffel* — попытку присваивания), чтобы иметь возможность работать с полученным элементом как с числом. В этом случае могло бы пригодиться выведенное утверждение о том, что контейнер всегда хранит только целые числа. Однако такая противоестественная для объектно-ориентированного программирования ситуация не может встретиться в *Eiffel*, где есть обобщенные классы. Если *Eiffel*-разработчик объявляет сущность с некоторым статическим типом, подразумевается, что к ней во время выполнения могут быть присоединены объекты любых его подтипов.

Другой аспект взаимодействия динамического вывода утверждений с наследованием, обсуждавшийся в главе 1, как раз имеет важное значение в контексте *Eiffel*. Речь идет о взаимосвязи точек программы в классах, связанных отношением наследования. Для краткости будем называть такую взаимосвязь *наследованием точек программы*.

Для механизма наследования в *Eiffel* справедлив принцип подстановочности, который реализуется, в частности, в правиле *переопределения утверждений* (или, по-другому, правиле *субконтрактов*). В соответствии с этим правилом класс-потомок в праве только ослаблять предусловия компонентов класса-предка и только усиливать постусловия и инвариант класса. Как и в языке спецификации *JML*, выполнение этого правила навязывается формальным путем: потомку не разрешается определять утверждения в их окончательном виде. Он может лишь доопределить соответствующие утверждения предка, при этом окончательный вид (так называемая *плоская форма*) утверждения строится из его частей, определенных в предке и в потомке, путем взятия их дизъюнкции в случае предусловий и конъюнкции в случае постусловий и инвариантов. Чтобы сделать это правило более явным для разработчика, введена необходимость

использовать для доопределения предусловий и постусловий специальные ключевые слова `require else` и `ensure then`.

Как показано в работе [20] при таком способе записи контрактов стандартная техника динамического вывода утверждений может приводить к противоречиям. В *Eiffel* это особенно нежелательно, поскольку истинность утверждений проверяется автоматически в процессе работы системы. Для выхода из ситуации необходимо применить подход, подобный описанному в работе [20].

Однако применять в точности этот подход не совсем корректно, так как формально правило субконтрактов в *Eiffel* отличается от правила, приведенного в главе 1 для *JML*. Пусть, как и раньше, имеются два класса A и B, из которых первый является предком, а второй — потомком. Как и раньше, класс A определяет компонент f, который переопределяется в классе B, причем спецификации компонентов выглядят следующим образом:

```
class A
  ...
  f is
    require P
    ensure Q
  end
end
```

```
class B
  ...
  f is
    require else R
    ensure then S
  end
end
```

Тогда по правилам *Eiffel* полное предусловие и постусловие компонента f в классе B будут выражаться формулами:

$$pre_{B.f} = P \vee R$$
$$post_{B.f} = P \vee R \implies Q \wedge S$$

Сравните последнюю формулу с соответствующей формулой для *JML*:

$$post_{B.f} = (P \implies Q) \wedge (R \implies S)$$

Правило субконтрактов *Eiffel* сильнее, чем соответствующее правило *JML*, в том смысле, что накладывает больше ограничений на реализацию потомка. Здесь переопределенный компонент обязан удовлетворить обе части постусловия вне зависимости от того, какая из частей предусловия была истинна перед началом его выполнения. Однако, хоть это и кажется странным на первый взгляд, реализовывать переопределенные компоненты по правилу *Eiffel* не

сложнее, а даже проще, так как нет необходимости проверять, какая из частей предусловия была выполнена. Кроме того, в этом случае проще осуществлять мониторинг утверждений во время выполнения системы.

Алгоритм вывода утверждений, преложенный в работе [20], содержал два прохода, поскольку в соответствии с правилом *JML*, не зная предусловий компонентов предка и потомка, невозможно определить, обязан ли компонент потомка в данном конкретном вызове соблюсти часть постусловия, определенную в предке. С правилом субконтрактов *Eiffel* такой проблемы не возникает: компонент потомка *всегда* должен соблюдать унаследованную часть постусловия. Поэтому правило динамического вывода постусловий, предложенное в работе [20], следует модифицировать следующим образом: *значения на выходе из полиморфного вызова используются для вывода постусловия его динамического получателя и всех компонентов, которые он переопределяет.* Таким образом, вывод утверждений можно выполнять в один проход. Отметим, что использование предлагаемого правила не требует модификации детектора утверждений, оно реализуется исключительно за счет оснастителя.

Поскольку в *Eiffel* использование множественного наследования является крайне распространенной практикой, следует оговорить одно уточнение правила вывода предусловий из работы [20]. В нем говорится, что значения на входах полиморфных вызовов используются для вывода предусловий всех компонентов, переопределяемых динамическим получателем, *вплоть до* статического получателя. В контексте множественного наследования это означает, что версии компонента выбираются из классов, которые принадлежат к *решетке*, образованной отношением наследования и ограниченной снизу и сверху классами статического и динамического получателя, соответственно.

Кроме того, в работе [20] ничего не сказано об инвариантах класса. Разумно сформулировать для инвариантов следующее правило: *значения на входах и выходах всех квалифицированных вызовов, динамические получатели которых определены в одном классе, используются для вывода инварианта этого класса и всех его предков.*

Отметим, что предлагаемая техника служит для вывода утверждений в их плоской форме. Для того, чтобы перейти к более удобной иерархической форме, предположительно можно использовать механизм *подавления* одних точек программы другими, существующий в детекторе *Daikon*. Если указать детектору,

что, например, стабильное состояние предка подавляет стабильное состояние потомка, то из выведенного инварианта потомка будут исключены те предложения, которые следуют из инварианта предка.

Приведем пример использования предложенных выше правил. Рассмотрим иерархию классов, изображенную на рис. 4. Здесь классы A и E независимо определяют компонент f, класс B переопределяет f из класса A, после чего класс C наследует его без изменений, а в классе D происходит слияние двух компонентов f из классов B и E.

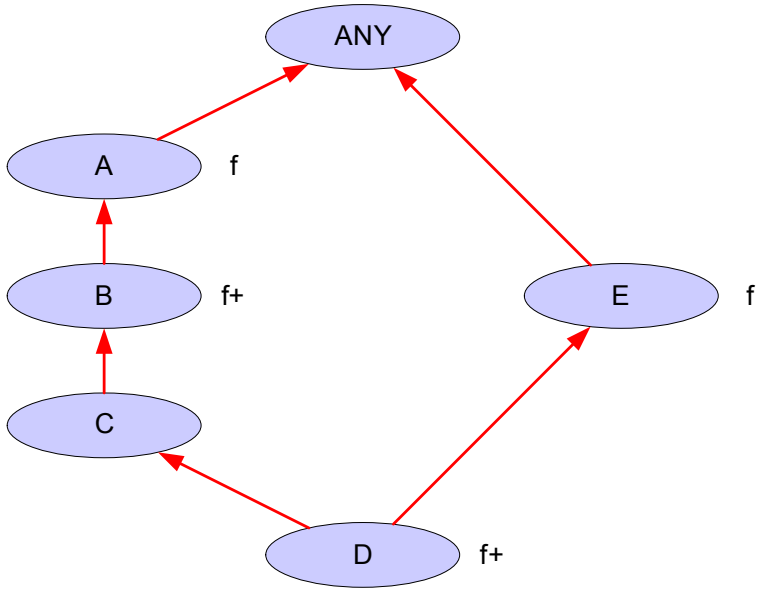

Рис. 4. Пример иерархии классов

Рассмотрим полиморфный вызов $c.f$, где сущность c объявлена с типом C, однако в момент вызова к ней присоединен объект типа D. Таким образом, динамическим получателем вызова является $D.f$, а статическим — $B.f$, так как в классе C нет версии этого компонента. В соответствии с описанными выше правилами состояние системы на входе в этот вызов будет использовано для вывода предусловий компонентов $D.f$ и $B.f$ (но не $A.f$ и $E.f$, так как классы A и E не принадлежат решетке наследования, ограниченной классами B и D), а также для вывода инвариантов всех изображенных на рисунке классов. Состояние на выходе из этого вызова будет использовано для вывода постусловий компонентов $D.f$, $B.f$, $A.f$ и $E.f$, а также инвариантов всех изображенных на рисунке классов.

Учет наследования точек программы при выводе утверждений позволяет не только избавиться от противоречий и получить концептуально правиль-

ные спецификации, но и выводить контракты для отложенных классов, поведение которых невозможно наблюдать непосредственно. Рассмотрим вымышленный пример с отложенным классом LIST (список) и двумя его реализациями LIST_ONE и LIST_TWO. Пусть список содержит внутренний курсор, запросы `before` и `after` являются индикаторами расположения курсора перед или после любого элемента списка соответственно. Пусть класс LIST_ONE реализован таким образом, что в случае пустого списка оба запроса возвращает значение «истина», а разработчик класса LIST_TWO, напротив, решил, что эти два запроса никогда не должны быть истинными одновременно, и в случае пустого списка следует возвращать «истину» по запросу `before` и «ложь» по запросу `after`. В инварианте LIST_ONE детектор утверждений мог бы вывести

```
is_empty implies before and after,
```

а в инварианте LIST_TWO

```
is_empty implies before and not after.
```

Поскольку для вывода инварианта отложенного класса LIST в соответствии с предложенным выше правилом будут использованы значения запросов `is_empty`, `before` и `after` в стабильных состояниях как объектов типа LIST_ONE, так и объектов типа LIST_TWO, детектор выведет лишь то свойство, которое является общим для обоих инвариантов, а именно

```
is_empty implies before.
```

Такой результат кажется логичным, и все же возникает разумное возражение: если бы во время выполнения системы создавались только объекты класса LIST_ONE, то инвариант, выведенный для него, был бы повторен без изменений в классе LIST, что некорректно. Это действительно так, однако это всего лишь проявление основного свойства динамического вывода утверждений: зависимости результатов вывода от входных данных. В этом смысле решение не учитывать наследование точек программы аналогично решению вообще отказаться от динамического вывода из-за риска получения неверных утверждений. Отметим, однако, что детектор *Daikon* производит статистическую оценку степени доверия выведенным утверждениям перед тем, как сообщить их пользователю.

Если вероятность того, что утверждение оказалось истинным просто по совпадению, больше определенной границы, это утверждение исключается из числа выведенных. Возможно, учет информации о потомках классов и переопределениях компонентов поможет более точно оценивать степень доверия выведенным утверждениям в условиях наследования точек программы.

2.6. Другие особенности

В языке *Eiffel* есть и другие интересные особенности по сравнению с языком *Java*, которые следует учитывать в процессе динамического вывода утверждений.

Одна из таких особенностей — обобщенные классы (ради справедливости следует отметить, что на данный момент обобщенные классы есть и в *Java*). Такие классы параметризованы и типы сущностей в их тексте могут совпадать с формальными параметрами. Поскольку класс, на котором основан тип сущности, в данном случае неизвестен, казалось бы, невозможно определить множество ее дочерних переменных. Однако в действительности это не представляет проблемы: по определению, данному в разд. 2.3.2, дочерние переменные порождаются запросами *базового класса* родительской переменной. Понятие «базовый класс» языка *Eiffel* очень удобно в данном случае. Базовый класс определен для любого типа, и в случае формального параметра совпадает с его *ограничением* (классом, от которого должны наследоваться все фактические параметры, соответствующие данному формальному). Напомним, что в случае неограниченной обобщенности ограничением по умолчанию считается класс `ANY`. Такое определение дочерних переменных корректно, так как множество компонентов, которые разрешается вызывать в обобщенном классе на сущностях с типом формального параметра, как раз и определяется его ограничением.

Другая особенность *Eiffel*, о которой следует упомянуть — это *однократные подпрограммы*. Если подпрограммы объявлена как однократная, она будет выполняться только однажды за все время работы системы, при первом вызове. Если однократная подпрограммы является функцией, ее результат запоминается и при каждом последующем вызове возвращается без вычисления. Однократные функции, так же, как и обыкновенные, могут отражать интересные свойства переменных. Однако использование однократных функций в процессе

развертывания может привести к тому, что первый вызов произойдет в ином контексте, чем в оригинальной системе, а следовательно, значение функции и поведение всей системы моет измениться. Одно из возможных решений этой проблемы состоит в том, чтобы во время выполнения оснащенной системы следить за тем, какие однократные функции уже были вызваны в их оригинальном контексте, а какие еще нет. Пока функция не была вызвана, ее значение записывается в файл истории как `nonsensical` (бессмысленное). Подобным образом поступает отладчик среды разработки *EiffelStudio*: в нем невозможно узнать значение однократной функции, пока оно не было вычислено в исходной системе. В контексте динамического вывода утверждений такое решение нельзя назвать безупречным, однако ничего лучшего пока не придумано.

Глава 3.

Инструментальное средство *CITADEL*

Предлагаемая техника динамического вывода утверждений в языке *Eiffel* реализована автором в инструментальном средстве *CITADEL* (Contract Inference Tool that Applies Daikon to Eiffel Language). Как можно заключить из названия, инструментальное средство представляет собой *Eiffel*-интерфейс к детектору утверждений *Daikon*.

3.1. Архитектура

Согласно описанию системы динамического вывода утверждений в главе 1, интерфейс к *Daikon* для любого языка программирования состоит из оснастителя и, возможно, построцессоров. Кроме того, в руководстве для разработчиков [12] при реализации поддержки нового языка спецификации рекомендуется модифицировать ту часть *Daikon*, которая отвечает за преобразование выведенный утверждений в текстовую форму, чтобы иметь возможность получать их сразу в нужном формате.

Основную сложность при разработке интерфейса представляет реализация *оснастителя*. Входными данными для оснастителя в *CITADEL* являются исходная программная система (в *Eiffel* система — это множество классов и специальный файл конфигурации), а также список тех кластеров и классов, которые необходимо обработать. Как было упомянуто выше, частичное оснащение системы позволяет избежать проблем с масштабируемостью. Оснаститель *CITADEL* выполняет следующие задачи.

1. Синтаксический анализ кода исходной системы и анализ типов.
2. Поиск в абстрактном синтаксическом дереве интересных точек программы.
3. Поиск видимых переменных в каждой интересной точке программы.

4. Развертывание переменных.

5. Создание файла объявлений, в котором для каждой интересной точки программы указано ее имя, имена и типы всех переменных и другая статическая информация.

6. Поиск в абстрактном синтаксическом дереве подходящих мест для внедрения инструкций записи значений переменных в файл истории.

7. Генерация этих инструкций (с проверками возможности вычисления значения переменной, в частности, с проверками предусловий функций).

8. Генерация кода, позволяющего получать идентификатор переменной во время выполнения (в частности, различать объекты печатаемых, ссылочных и развернутых типов).

9. Генерация кода, позволяющего следить за статусом однократных функций.

Выходными данными оснастителя являются файл объявлений точек программы и *оснащенная система*, состоящая из оснащенных классов исходной системы, вспомогательных классов, реализующих работу с файлом истории, идентификаторами и однократными функциями, и нового файла конфигурации.

Далее *CITADEL* компилирует и запускает оснащенную систему. В процессе ее работы создается файл истории. Два файла: объявлений и истории — подаются на вход детектору *Daikon*. Детектор был модифицирован таким образом, что после вывода утверждений он преобразует их в текстовый вид уже в формате *Eiffel*.

После этого свою работу начинает *аннотатор*, входными данными которого являются текстовый файл с выведенными утверждениями и снова та же самая исходная система. Основная задача аннотатора — найти в абстрактном синтаксическом дереве исходной системы те места, в которые необходимо вставить выведенные утверждения. Кроме того аннотатор осуществляет некоторую пост-обработку этих утверждений, необходимость которой обусловлена, в основном, несовершенством детектора *Daikon*. При выводе утверждений *Daikon* просто-напросто игнорирует те наблюдения переменной, где она имела значение `nonsensical`, и в результате утверждения с участием частичных перемен-

ных[1] становятся неверными. Например, для вещественной переменной `x`, принимавшей во время выполнения как неотрицательные, так и отрицательные значения, *Daikon* может вывести свойство `x.sqrt >= 0`, где функция `sqrt` определена только для неотрицательных значений `x`. Такие утверждений можно сделать верными (и релевантными), если добавить к ним *охраняющее условие*, при котором все частичные переменные определены. В приведенном выше примере выведенное утверждение следовало бы преобразовать в

```
x >= 0 implies x.sqrt >= 0.
```

Детектор *Daikon* мог бы выводить такие охраняющие условия (у него есть функция вывода условных утверждений), однако, по каким-то причинам, он этого не делает. Один из способов решения этой проблемы состоит в постобработке выведенных утверждений. Аннотатор *CITADEL* просматривает файл истории и находит в нем все частичные переменные. После этого к каждому утверждению, содержащему частичную перменную, аннотатор добавляет охраняющее условие, совпадающее с условием возможности вычисления значения этой переменной, которое было определено в процессе оснащения системы.

Результатом работы аннотатора (и всего инструмента *CITADEL*) является аннотированный код классов, выбранных для обработки в исходной системе.

Для синтаксического анализа и анализа типов исходной системы инструмент *CITADEL* использует библиотеку *Gobo Tools* [21]. Компиляция оснащенной системы осуществляется с помощью компилятора от *Eiffel Software*, входящего в поставку среду разработки *EiffelStudio*.

В заключение рассмотрим структуру основных кластеров системы *CITADEL*.

Program point. В этом кластере содержится иерархия разновидностей точек программы (рис. 5). От главного отложенного класса «точка программы» наследуются «стабильное состояние», в котором множество видимых переменных состоит только из текущего объекта `Current`, и «точка подпрограммы», в которой к числу видимых переменных добавляются еще и формальные аргументы подпрограммы. От этого класса наследуются

[1] Будем называть *частичными* для некоторого выполнения оснащенной системы те переменные, значения которых во время этого выполнения не всегда были определены

«вход в подпрограмму», «выход из подпрограммы» и «точка в теле подпрограммы» (в ней видимыми являются еще и локальные переменные). Ортогональным трем перечисленным выше альтернативам является класс «точка функции», в котором к числу видимых переменных добавляется результат функции `Result`. У входа в подпрограмму есть специальный случай — «вход в процедуру создания», в котором отсутствует переменная `Current`. От точки в теле подпрограммы наследуется точка «цикл». Точка выхода и точка функции вместе образуют «точку выхода из функции», при этом их множества видимых переменных объединяются. Точка в теле подпрограммы и точка функции порождают «точку в теле функции», которая совместно с точкой «цикл» образует «цикл внутри функции».

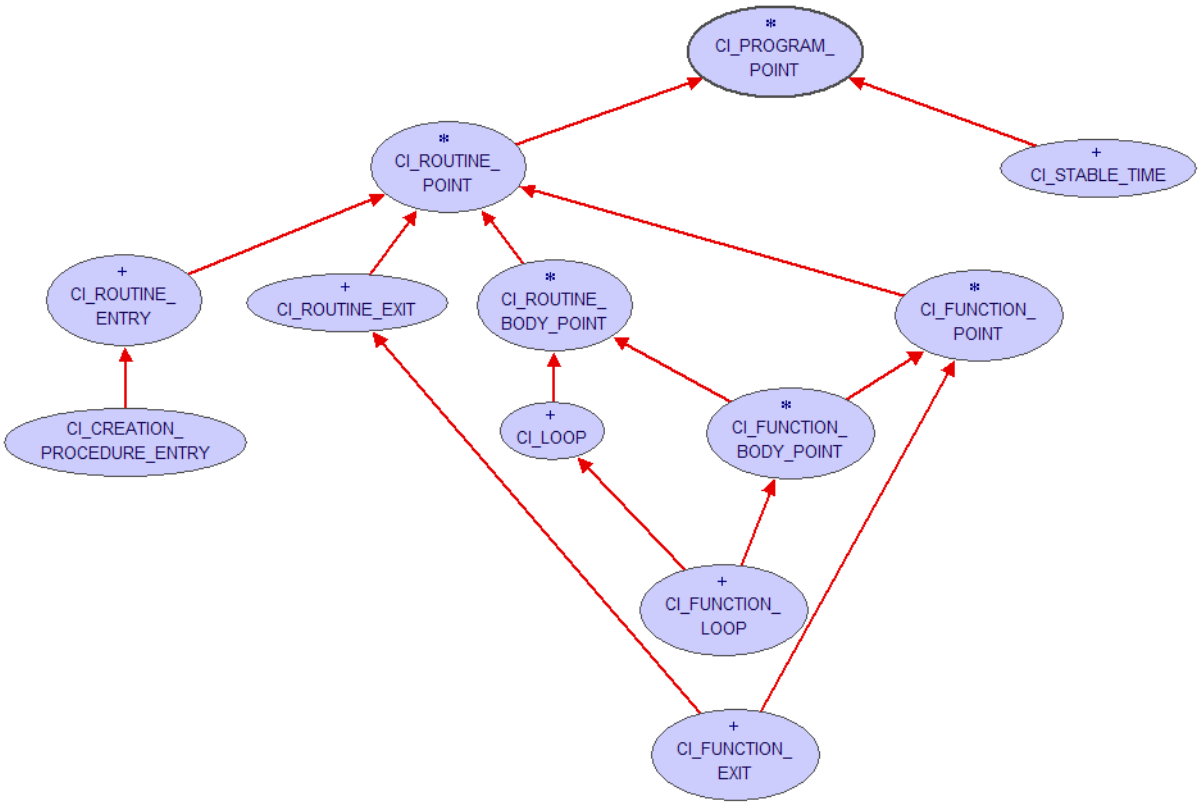

Рис. 5. Иерархия точек программы в *CITADEL*

Variable. Этот кластер содержит иерархию переменных (рис. 6). Здесь от главного отложенного класса «переменная» наследуются различные виды переменных: «текущий объект», «формальный аргумент», «локальная переменная», «результат функции», «запрос» (к этому виду принадлежат все

дочерние переменные) и «значение строки» (специальный вид дочерней переменной, который обсуждался в разд. 2.3.2).

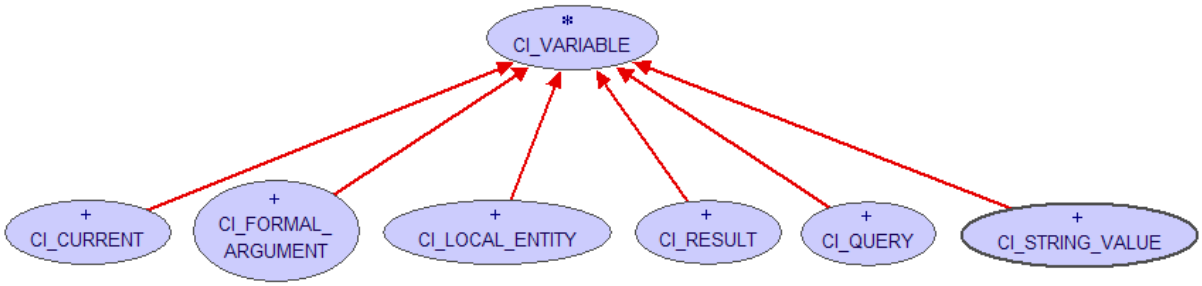

Рис. 6. Иерархия переменных в *CITADEL*

Variable type. Здесь находятся все классы, связанные с типами переменных (рис. 7). Класс «печатаемые типы» содержит информацию о печатаемых типах *Daikon*. Класс «тип переменной» инкапсулирует информацию об объявленном типе переменной: имя типа, его базовый класс, его *вид* в смысле правила вычисления идентификатора переменной. Для хранения вида предназначен специальный отложенный класс, от которого наследуются конкретные виды: «печатаемый», «ссылочный», «развернутый» и «неизвестный». Последний вариант используется в том случае, когда вид необходимо определить динамически, во время исполнения оснащенной системы. В разд. 2.3.1 утверждалось, что правило вычисления идентификатора всегда определяется динамическим типом переменной, однако во многих случаях определить вид возможно на этапе оснащения, и в целях повышения быстродействия *CITADEL* так и поступает. Например, если объявленный тип переменной является развернутым (по правилам *Eiffel* развернутые классы не могут иметь согласованных потомков, иными словами, сущность развернутого типа не является полиморфной) или тип является ссылочным, однако его базовый класс не содержит ни развернутых, ни печатаемых потомков.

Processing. Этот кластер объединяет такие высокоуровневые понятия, как «оснастителъ» и «аннотатор» (рис. 8). Эти два класса наследуются от общего отложенного родителя, в котором реализуется функциональность поиска точек программы, а определение способа их обработки оставлено на усмотрения потомков. Классы «оснастителъ» и «аннотатор» реализуют

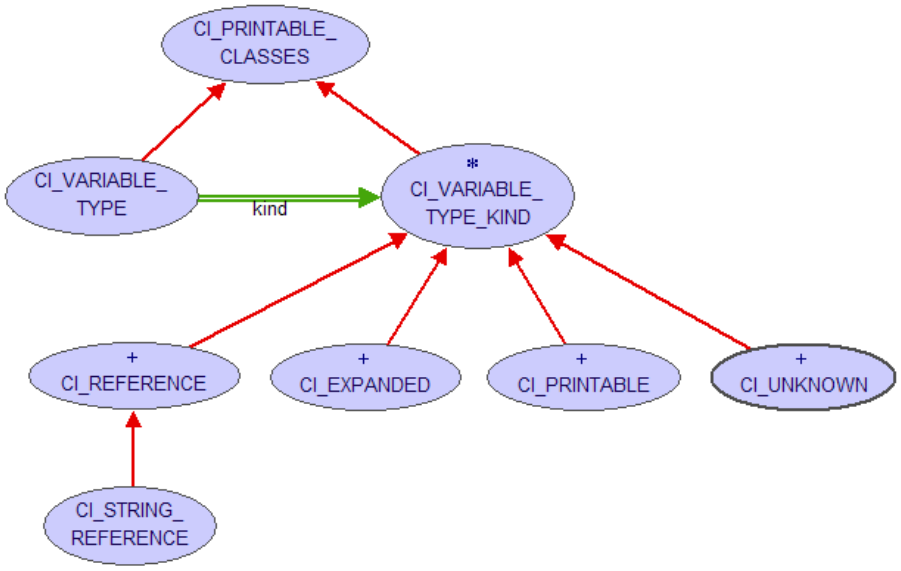

Рис. 7. Информация о типах переменных в *CITADEL*

эту обработку, каждый по-своему. Функции формирования статической и динамической информации о точках программы инкапсулированы в отложенном классе «принтер точек программы». Его эффективному наследнику по имени «принтер точек программы для *Daikon*» известен конкретный формат этой информации, специфичный для детектора утверждений *Daikon*.

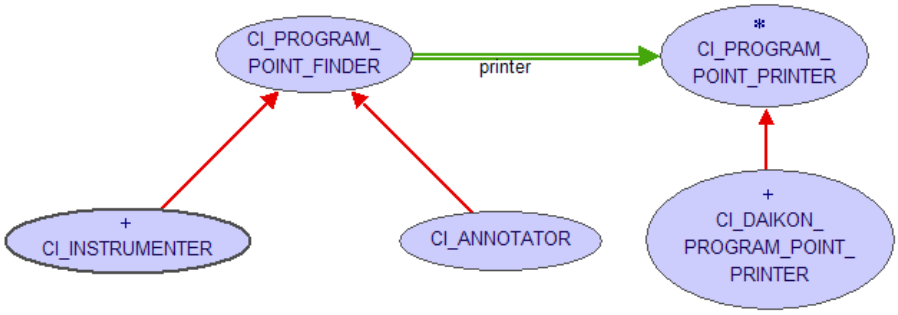

Рис. 8. Высокоуровневые классы *CITADEL*

3.2. Особенности реализации

Поддержка проектирования по контракту в языке *Eiffel* позволяет применить интересную технику при разработке оснастителя. Реализация функции внедрения в интересные точки программы инструкций вывода значений переменных порождает три основные проблемы:

- В какие места кода внедрить инструкции, чтобы вывод производится в нужный момент времени? Ответить на этот вопрос довольно просто в случае предусловий (в начале тела подпрограммы), постусловий (в конце тела подпрограммы) и инвариантов циклов (в конце составных инструкций `from` и `loop`). Однако решение не так очевидно в случае инварианта класса. Как было упомянуто в главе 2 стабильные состояния объекта наблюдаются только во время квалифицированных вызовов. Отслеживание квалифицированности вызовов потребовало бы серьезных модификаций всей оснащаемой системы.

- Если при развертывании используется функция оснащаемого класса, должна вызываться ее оригинальная, а не оснащенная версия. Отметим, что этого эффекта можно добиться с помощью глобального флага. Значение флага должно проверяться перед каждым блоком инструкций записи состояния. Если флаг сброшен, то он устанавливается, затем выполняется блок инструкций вывода и флаг снова сбрасывается. Если флаг уже установлен, инструкции вывода не выполняются.

- Реализация правил, описанных в разделе разд. 2.5, для поддержки наследования точек программы тоже может оказаться нетривиальной в виду разнообразия возможностей, связанных с наследованием в языке *Eiffel*.

Все три проблемы удалось решить разом, поместив инструкции записи состояния системы в каждой точке программы внутрь утверждения, соответствующего этой точке. Конечно, утверждения не могут в явном виде содержать инструкций, поэтому каждой точке программы потребовалось сопоставить булеву функцию, которая осуществляет запись состояния системы и возвращает значение «истина». Обычно в *Eiffel* не следует использовать функции с побочным эффектом, но в коде, который генерируется автоматически и не предназначен для повторного использования это вполне допустимо.

Первая проблема из приведенного выше списка решается по той причине, что запись состояния системы производится именно в том месте, где его свойства, выведенные детектором утверждений, в последствии будут проверяться (а значит там они и должны выполняться). Вторая проблема решается за счет того, что по правилам *Eiffel*, если утверждение содержит вызов функции, то при проверки этого утверждения во время выполнения программной системы,

контракты функции уже не проверяются. Что касается третьей проблемы, она решается, поскольку мониторингу во время выполнения подвергаются плоские формы контрактов. Однако, как будет пояснено ниже, здесь не все так просто, если речь идет о точках входа в подпрограмму.

Кроме перечисленных выше трех проблем, предлагаемый подход решает вопрос вывода утверждений для отложенных и внешних (реализованных на другом языке программирования) компонентов: ведь у них нет тела, и при обычном подходе инструкции записи состояния было бы некуда поместить, однако снабдить контрактами такие компоненты можно.

К сожалению, предлагаемый подход не работает в контексте наследования точек программы типа «вход». Такие точки соответствуют предусловиям, плоские формы которых, в отличие от других утверждений, представляют собой дизъюнкцию, а не конъюнкцию частей, определенных в разных классах. Эта специфика приводит к тому, что при одном и том же полиморфном вызове *выводиться* должны предусловия всех версий компонента от статического до динамического получателя, а *проверяться* — предусловия всех версий, начиная с самых ранних, до статического получателя (причем в любом порядке и, возможно, не все, а только до тех пор, пока один из дизъюнктов не окажется истинным). Более того, в среде разработки *EiffelStudio* мониторинг предусловий реализован не совсем корректно: при полиморфном вызове проверяются части предусловия из всех версий вплоть до динамического получателя (а не статического, что было бы правильнее), в результате чего некоторые нелегальные вызовы не обнаруживаются.

Поэтому придется отказаться от записи состояния системы в предусловиях и перенести их на сторону клиента, поместив перед вызовом, внутрь проверки. Поддержку наследования точек типа «вход» необходимо будет реализовать вручную. Конечно, запись состояния на стороне клиента неудобна, так как требует модификации всей системы, а не только оснащенных классов. Однако, по-видимому, без этого обойтись не удастся, так как информацией о статическом получателе вызова, необходимой для корректного вывода предусловий, располагает только клиент.

Вспомним пример иерархии классов из разд. 2.5 (рис. 4) и рассмотрим на этот раз полиморфный вызов $a.f$, где сущность a определена с типом A, но во время вызова присоединена к объекту типа D. При использовании пред-

лагаемого подхода код вызова на стороне клиента видоизменится следующим образом:

```
...
check a.a_f_pre end
a.f
...
```

Здесь функция `a_f_pre` сгенерирована оснастителем и определена, вместе с компонентом f, в классе A. В классах B и D, переопределяющих f, определены, соответственно, функции `b_f_pre` и `d_f_pre`. Код функций `a_f_pre` и `b_f_pre` должен выглядеть приблизительно следующим образом:

```
a_f_pre: BOOLEAN is
  local
    b: B
  do
    -- Печать переменных точки
       входа в A.f
    ...
    -- Поддержка наследования
       точек входа
    b ?= Current
    if b /= Void then
      b.b_f_pre
    end
  end
```

```
b_f_pre: BOOLEAN is
  local
    d: D
  do
    -- Печать переменных точки
       входа в B.f
    ...
    -- Поддержка наследования
       точек входа
    d ?= Current
    if d /= Void then
      d.d_f_pre
    end
  end
```

Реализация концепции наследования точек программы влечет еще одну проблему. Детектор *Daikon* предоставляет крайне удобную возможность вывода постусловий с участием old-выражений без необходимости явного введения соответствующих им переменных. Для этого детектор ведет учет соответствия входов и выходов из каждой подпрограммы: в качестве исходного значения некоторой переменной на выходе используется значение переменной с тем же именем на входе. Однако в предлагаемой модели наследования точек программы не каждому выходу из подпрограммы обязательно соответствует вход, так как множество версий компонента, соответствующих выходу из полиморфного вызова, в общем случае более широко, чем множество версий, соответствующих входу. По-видимому, единственное решение этой проблемы — отключить

слежение за соответствием входов и выходов в детекторе и в явном виде генерировать переменные, соответствующие old-выражениям, в интерфейсе. При этом может возникать ситуация, когда вход в подпрограмму не наблюдался, но значение old-выражений на ее выходе может быть вычислено. Эта ситуация может показаться противоречивой, однако правил языка она не нарушает. Это уникальный случай, когда исходные значения переменных, даже не удовлетворяя предусловию компонента, тем не менее могут быть использованы для вывода его предусловия.

3.3. Ограничения

Перечислим функциональность, которая пока не реализована в интерфейсе *CITADEL*, но может быть полезна при динамическом выводе утверждений в *Eiffel*.

- Использование функций с аргументами при развертывании. Эта возможность обсуждалась в разд. 2.3.2 и разд. 2.4. На данный момент в процессе развертывания участвуют только атрибуты и функции без аргументов.

- Полнофункциональное наследование точек программы. В текущей версии поддерживается только наследование стабильных состояний. Наследование точек входов не поддерживается из-за сложностей реализации, упомянутых в предыдущем разделе. Наследование точек выходов не используется из-за проблем с учетом соответствия входов и выходов в детекторе.

- Анализ сравнимости переменных. Многие нерелевантные утверждения, выведенные детектором *Daikon*, представляют собой сравнения несравнимых переменных, например «число детей человека больше года его рождения». Для языка *Java* существует инструмент, который производит анализ исходного кода и распределяет переменные по классам сравнимости путем применения ряда эвристик. Возможно, подобный инструмент необходимо реализовать и для *Eiffel*.

- Поддержка инкрементального режима вывода утверждений. Такой режим может понадобиться при обработке большой иерархии классов, когда проблемы масштабируемости дадут о себе знать.

- Поддержка параллелизма. Запись истории выполнения параллельных программ требует особых техник, которые на данный момент не реализованы.

Глава 4.

Экспериментальная проверка

Располагая техникой динамического вывода утверждений в языке *Eiffel* и инструментальным средством *CITADEL*, наконец можно перейти к основному вопросу настоящей работы, а именно, к исследованию применимости вывода утверждений в контексте проектирования по контракту вообще и в языке *Eiffel* в частности.

Совместно с коллегой из Высшей Политехнической Школы в Цюрихе автором был проведен эксперимент с целью выяснить, действительно ли вывод утверждений с использованием детектора *Daikon* и интерфейса *CITADEL* может быть использован для тех целей, которые освещались в разд. 2.1: усиления и исправления контрактов, улучшения качества тестовых наборов. Кроме того, интересно было оценить качество выведенных контрактов, сравнить их с контрактами разработчика и, в конце концов, ответить на вопрос, следует ли перепоручить создание формальных спецификаций, или в этом вопросе приоритет пока остается за человеком.

4.1. Выбор классов

В процессе эксперимента вывод утверждений производился для отдельных классов, поэтому проблемы масштабируемости и вопросы наследования точек программы роли не играли. В то же время, эксперимент проводился не на искусственных примерах, а на существующих классах, созданных сторонними разработчиками не для целей вывода утверждений. Эти классы никак не адаптировались и не модифицировались в процессе эксперимента.

Выбор классов был нарочно сделан из двух полярно противоположных областей ПО: первая группа классов была выбрана из библиотек *Base* и *Gobo*, а вторая — из курсовых работ студентов Высшей Политехнической Школы.

Библиотеку *Base* можно смело назвать стандартной библиотекой *Eiffel*: она поставляется вместе с наиболее популярной средой разработки *EiffelStudio*

и используется во всех программных системах, разработанных в этой среде. В ней содержатся такие «краеугольные» классы, как ANY, INTEGER и т.п., а также основные структуры данных и другие классы «первой необходимости». Вторая из упомянутых библиотек — *Gobo* — является почти стандартной: она также поставляется вместе с *EiffelStudio* и используется очень широко. *Gobo* предоставляет расширенные возможности работы со структурами данных, строками *Unicode*, формальными языками и многим другим. Особенность классов, содержащихся в этих двух библиотеках, в том, что уровень их повторного использования крайне велик. Следовательно, велики и затраты на обеспечение их качества, в частности они, как правило, снабжены проработанными и, по возможности, полными контрактами.

Напротив, для классов из студенческих проектов повторное использование крайне маловероятно. Поэтому и качество контрактов в них предположительно низкое (это, конечно, не всегда так, поскольку часто спецификация разрабатывается преподавателем и служит для студента заданием на реализацию класса).

Для полноты картины следовало бы еще включить в эксперимент несколько классов из области промышленного ПО. Однако, процесс оценки выведенных утверждений достаточно трудоемкий и неоднозначный, особенно когда семантика класса не вполне прозрачна. Поэтому для работы с промышленными классами, которые часто имеют большой размер и сложную семантику, необходимо подключать их авторов. Такой возможности в процессе проведения эксперимента не было.

Для эксперимента было выбрано семь классов из библиотек и четыре — из студенческих проектов. Эти классы разного размера, с различной глубиной наследования и с разнообразной, но ясной семантикой. Приведем краткое описание каждого из выбранных классов.

- BASIC_ROUTINES. Библиотека: *Base*. Размер: 32 LOC[1].

  Это небольшой класс предоставляющий доступ к некоторым простым операциям с числами (абсолютное значение числа, его знак и т.п.). Пример результатов выода утверждений для этого класса приведен в приложении 1.

- COMPARABLE. Библиотека: *Base*. Размер: 117 LOC.

[1]Размер класса здесь измеряется в строках кода (lines of code, LOC)

Это отложенный класс, представляющий концепцию элемента полностью упорядоченного множества. Особенность этого класса в том, что все его контракты принадлежат к высокому уровню абстракции: это аксиомы, отражающие связь различных отношений порядка между собой, и свойства их рефлексивности и симметричности.

- BOOLEAN_REF. Библиотека: *Base*. Размер: 174 LOC.

  Этот класс описывет ссылку на логическое значение. В нем определяется все стандартные булевы операции (такие как «и», «или», «не»), но не напрямую, а в терминах операций над хранимым значением.

- ST_SPLITTER. Библиотека: *Gobo*. Размер: 389 LOC.

  Класс, предоставляющий функции разбиения слияния строк с произвольными символами-разделителями и escape-символами. Особенности этого класса: работа со строками и очень высокое качество контрактов разработчика.

- TIME. Библиотека: *Base*. Размер: 401 LOC.

  Класс реализует концепцию абсолютного времени с операциями получения и изменения часов, минут, секунд и дробной части секунды. Особенность этого класса: множество целочисленных запросов, приводящее к большому объему выведенных утверждений.

- INTEGER_INTERVAL. Библиотека: *Base*. Размер: 469 LOC.

  Класс представляет собой специфический вид контейнера — интервал целых чисел.

- DS_TOPOLOGICAL_SORTER. Библиотека: *Gobo*. Размер: 487 LOC.

  Этот класс реализует операцию топологической сортировки и предоставляет связанные с ней сервисы.

- FRACTION_1. Студенческий проект. Размер: 166 LOC.

  Класс реализует концепцию рационального числа. В задании, данном студентам, требовалось, чтобы класс был потомком NUMERIC (элемент коммутативного кольца) и определял все его операции, а также навязывалось

свойство отсутствие у числителя и знаменателя общих делителей (сокращенность дроби).

- FRACTION_2. Студенческий проект. Размер: 156 LOC.

  Этот класс является другим решением той же задачи. Пример результатов выода утверждений для этого класса приведен в приложении 1.

- GENEALOGY_1. Студенческий проект. Размер: 874 LOC.

  Класс является реализацией генеалогического дерева с множеством операций над персоналиями и разнообразными отношениями между ними. Класс обеспечивает соблюдение большого числа хронологических и прочих ограничений (например, запрет многоженства).

- GENEALOGY_2. Студенческий проект. Размер: 1501 LOC.

  Этот класс является другим решением той же задачи.

4.2. Методика проведения эксперимента

Для выбранных классов автором был разработан набор модульных тестов. В работе [10] утверждается, что модульные тесты гораздо менее пригодны для целей вывода утверждений, чем системные. Однако в этой работе модульные тесты понимаются очень ограничено: как тесты, испытывающие небольшое число крайних случаев поведения программного модуля. Конечно, в таком случае результаты динамического вывода не могут быть хорошими, так как вывод предполагает наблюдение типичного поведения программы, причем число наблюдений должно быть достаточным для формирования статистики. Модульные тесты в настоящей работе испытывают как типичные, так и крайние случаи поведения класса. Работать с модульными тестами (в частности, производить измерения) гораздо проще и удобнее, чем с системными.

Для каждого класса вывод утверждений осуществлялся на трех различных тестовых наборах: маленьком, среднем и большом. Маленький набор строился методом случайного тестирования, испытывал лишь наиболее типичное поведение класса и содержал (примерно) по десять вызовов каждой подпрограммы класса. В средний набор кроме типичного поведения было включено испытание различных крайних случаев, и он содержал по 50 вызовов каждой

подпрограммы класса. Большой набор был качественно эквивалентен среднему, но содержал по 500 вызовов каждой подпрограммы.

На основании экспериментальных данных требовалось подтвердить или опровергнуть следующие теоретические предположения.

П1 Динамический вывод утверждений может быть использован для усиления контрактов разработчика. Более конкретно, если из выведенных утверждений исключить «общую» с утверждениями разработчика часть, останется еще много релевантных предложений. Кроме того, существует достаточно большое число точек программы, в которых утверждение разработчика вообще отсутствует, а детектору *Daikon* удается вывести что-нибудь релевантное.

П2 Подавляющее большинство релевантных выведенных предложений имеют одну из трех форм:

$$x \sim y \tag{1}$$

$$\texttt{Result} = b \implies a \tag{2}$$

$$x = 0 \implies z = 0 \tag{3}$$

где x, z — перменные, y — переменная или константа, $\sim$ — одно из отношений $=$, $\neq$, $<$, $\leqslant$, $>$, $\geqslant$, b — логическая константа («истина» или «ложь»), a — утверждение вида (1). На данный момент в детекторе *Daikon* вывод условных утверждений (импликаций) применяется в ограниченном варианте: в качестве антецедента используется только переменная `Result` на выходе из булевых функций (утверждение вида (2)). Кроме того, существует специальный шаблон `ZeroTrack`, порождающий утверждения вида (3).

В то же время, большинство предложений разработчика, не выведенных *Daikon*, содержат или функции с одним аргументом, или импликации более общего вида, чем (2) и (3).

Проверка этого предположения важна для будущего усовершенствования *Daikon* и *CITADEL*. Первая часть позволит отказаться от лишних шаблонов утверждений в *Daikon*, а вторая послужит основанием для реализации поддержки функций с одним аргументом при развертывании в *CITADEL* и более широкого использования условных утверждений в *Daikon*.

П3 Техника сведения предусловия, описанная в разд. 2.1.1, может использоваться для улучшения качества тестовых наборов и поиска программных ошибок.

Выведенные утверждения (ВУ) сравнивались с плоской формой утверждений разработчика (УР). Инварианты циклов оценивались отдельно, поскольку, как было упомянуто выше, они считаются менее интересными. Предложения УР классифицировались по трем категориям:

- не выразимые в грамматике *Daikon* (среди них отдельно выделялись предложения, содержащие функции с одним аргументом и импликации);
- выразимые, но не следующие из выведенного утверждения;
- выразимые и следующие из выведенного утверждения.

Предложения ВУ можно поделить на следующие классы:

- неверные (являющиеся свойствами тестового набора);
- верные неинтересные (избыточные, не несущие информации, лишенные смысла: например, сравнения между несравнимыми переменными);
- чрезмерная спецификация (верные и интересные предложения, которые, однако не должны быть частью контракта, поскольку чересчур привязаны к реализации);
- релевантные и следующие из утверждения разработчика;
- релевантные и не следующие из утверждения разработчика.

Однако не все эти классы были действительно использованы в эксперименте. Во-первых, для определения чрезмерной спецификации не существует формальных критериев, поэтому потенциальные члены этого классы были отнесены к релевантным предложениям. Во-вторых, для простоты считалось, что множество предложений ВУ, следующих из УР, совпадает с множеством предложений УР, следующих из ВУ. Это было бы верно, если бы речь шла не о следовании, а об эквивалентности. В данном случае это лишь приближение.

Кроме того, в процессе эксперимента подсчитывалось число точек программы, в которых не было утверждения разработчика, но присутствовало релевантное выведенное утверждение. Также велся подсчет числа релевантных и нерелевантных выведенных утверждений, не имеющих вид (1), (2) или (3). В табл. 4 приведены описания всех величин, которые непосредственно измерялись в процессе эксперимента.

Таблица 4. Непосредственно измеряемые величины

Код	**Описание**
Dev	Число предложений разработчика
E	Число предложений разработчика, невыразимых в грамматике *Daikon*
1-arg	Число невыразимых предложений разработчика, содержащих функции с одним аргументом
impl	Число невыразимых предложений разработчика, содержащих импликации
C	Число выразимых предложения разработчика, не следующих из выведенного утверждения
Inf	Число выведенных предложений
F	Число неверных выведенных предложений
U	Число верных неинтересных выведенных предложений
R	Число релевантных выведенных утверждений особого вида (не совпадающего с (1), (2) или (3)
IR	Число нерелевантных выведенных утверждений особого вида
P	Число точек программы, где нет утверждения разработчика, но присутствует релевантное выведенное утверждение

На основе величин, измеряемых непосредственно, были вычислены более информативные производные величины. Описание вспомогательных производных величин содержится в табл. 5. Табл. 6 описывает производные величины, характеризующие ВУ безотносительно УР. В табл. 7 собраны величины, которые являются сравнительными характеристиками ВУ и УР. Наконец, табл. 8

описывает величины, назначение которых — оценка возможностей путей улучшения существующих инструментов вывода утверждений.

Таблица 5. Вспомогательные производные величины

Код	**Формула**	**Описание**
InfR	Inf − F − U	Число релевантных выведенных предложений
DevI	Dev − E − C	Число предложений разработчика, следующих из ВУ
InfD	Dev − E − C	Число выведенных предложений, следующих из УД (по соглашению равно DevI)

Таблица 6. Производные величины, характеризующие ВУ

B1	$\frac{\text{Inf} - \text{F}}{\text{Inf}}$	*Точность*: доля верных предложений среди всех выведенных предложений
B2	$\frac{\text{InfR}}{\text{Inf}}$	*Релевантность*: доля релевантных предложений среди всех выведенных предложений

4.3. Результаты эксперимента

В табл. 9 приведены результаты измерения основных производных величин для каждого из библиотечных в отдельности. Эти результаты не учитывают инварианты циклов. В табл. 10 те же результаты приведены для классов из студенческих проектов.

Данные в табл. 9 подтверждают теоретическое предположение о том, что для класса `COMPARABLE` процент выразимых утверждений (B3) и абсолютная полнота выведенных утверждения (B5) довольно низки по причине высокого уровня абстракции класса. Это идеальный случай для того, чтобы проверить, насколько использование функций с аргументами при развертывании может повысить уровень абстракции и полноту выводимых утверждений. Автором было вручную произведено оснащение класса `COMPARABLE`, с подключением к процессу развертывания функций с одним аргументом. Результаты вывода утвержде-

Таблица 7. Производные величины, характеризующие ВУ по сравнению с УР

B3	$\frac{\text{Dev} - \text{E}}{\text{Dev}}$	Доля выразимых предложений среди всех предложений разработчика
B4	$\frac{\text{DevI}}{\text{Dev} - \text{E}}$	*Выразимая полнота*: доля выразимых предложений, следующих из ВУ, среди всех выразимых предложений
B5	$\frac{\text{DevI}}{\text{Dev}}$	*Абсолютная полнота*: доля выразимых предложений, следующих из ВУ, среди всех предложений разработчика
B6	$\frac{\text{InfR} + \text{Dev} - \text{DevI}}{\text{Dev}}$	*Коэффициент усиления УР*: во сколько раз объединенный набор предложений больше множества предложений разработчика
B7	$\frac{\text{Dev} + \text{Dev} - \text{InfD}}{\text{InvR}}$	*Коэффициент усиления ВУ*: во сколько раз объединенный набор предложений больше множества релевантных выведенных предложений
B8	$\frac{\text{P}}{2r + l + 1}$	Доля точек программы, где нет УР, но есть ВУ (здесь r — число подпрограмм класса, l — число циклов в классе).

ний для этого случая приведены в табл. 11. Как следует из таблицы, использование функций с одним аргументом позволило достичь процента выразимых утверждений и максимальной абсолютной полноты в 48% против 33% без применения этой техники.

Наибольший интерес представляют табл. 12 и табл. 13, в которых собраны значения производных величин (без учета инвариантов циклов), усредненные по всем классам, участвовавшим в эксперименте. Величины, собранные в первой таблице, в отличие от собранных во второй не зависят от размера тестового набора. На основании этих данных можно делать выводы об истинности сделанных предположений и качестве утверждений, выводимых с помощью инструмента *CITADEL*.

Для того, чтобы дать некоторое представление о ситуации с инвариантами цикла, приведем табл. 14, где собраны значения тех же величин, что и в табл. 13, но вычисленные только для этого вида утверждений. Эксперимен-

Таблица 8. Производные величины, характеризующие возможности улучшения ВУ

B9	$\frac{\text{1-arg}}{\text{E}}$	Доля предложений, содержащих функции с одним аргументом, среди всех невыразимых предложений.
B10	$\frac{\text{impl}}{\text{E}}$	Доля предложений, содержащих импликации, среди всех невыразимых предложений.
B11	$\frac{\text{R} + \text{IR}}{\text{Inf}}$	Доля предложений особого вида среди всех выведенных предложений.
B12	$\frac{\text{R}}{\text{InfR}}$	Доля релевантных предложений особого вида среди всех релевантных выведенных предложений.

тальные данных подтверждают теоретические предположения о том, что для инвариантов циклов велики значения выразимой полноты, коэффициента усиления УР, а также доли тех циклов, в которых нет утверждений разработчика, но присутствует релевантный выведенный инвариант (B8). Все это объясняется тем, что разработчики пишут инварианты циклов гораздо реже, чем другие утверждения.

Оценив усредненные значения производных величин, можно сделать следующие *заключения.*

З1 Динамический вывод утверждений действительно можно использовать для усиления контрактов. Показатель: $\text{B6} > 100$.

З2 Динамический вывод утверждений на данный момент не способен полностью заменить усилия разработчика по созданию спецификаций. Показатель: $\text{B5} < 100$.

З3 Выведенные спецификации в большей степени усиливают контракты разработчика, чем наоборот. Показатель: $\text{B6} > \text{B7}$.

З4 Чуть реже, чем в одном из пяти случаев, релевантные выведенные утверждения встречаются в точках программы, где разработчик не отметил никаких свойств. Показатель $\text{B8} \approx 20$.

Таблица 9. Результаты измерения основных производных величин для библиотечных классов (в процентах)

Набор тестов	B1	B2	B3	B4	B5	B6	B7	B8
BASIC_ROUTINES								
Маленький	67	67	43	0	0	157	275	0
Средний	100	100	43	67	29	171	109	23
Большой	100	100	43	67	29	200	108	38
COMPARABLE								
Маленький	100	100	33	29	10	105	733	0
Средний	100	100	33	100	33	48	100	0
Большой	100	71	33	100	33	48	100	0
BOOLEAN_REF								
Маленький	77	73	82	97	79	235	110	3
Средний	100	94	82	97	79	310	101	3
Большой	100	93	82	97	79	361	101	3
ST_SPLITTER								
Маленький	48	46	62	92	56	222	125	3
Средний	93	91	62	97	59	356	100	3
Большой	92	91	62	92	56	344	102	3
TIME								
Маленький	19	19	83	71	59	1407	103	30
Средний	40	40	83	85	71	1944	101	30
Большой	47	47	83	85	71	1922	101	30
INTEGER_INTERVAL								
Маленький	8	3	83	93	77	41	225	16
Средний	61	51	83	99	82	185	100	16
Большой	67	53	83	99	82	190	100	16
DS_TOPOLOGICAL_SORTER								
Маленький	38	38	69	92	56	222	125	7
Средний	84	84	69	97	59	356	100	14
Большой	97	97	69	92	56	344	102	14

Таблица 10. Результаты измерения основных производных величин для классов из студенческих проектов (в процентах)

Набор тестов	B1	B2	B3	B4	B5	B6	B7	B8
FRACTION_1								
Маленький	83	79	95	88	85	162	110	19
Средний	97	51	95	100	97	172	100	19
Большой	97	52	95	100	97	170	100	19
FRACTION_2								
Маленький	93	92	97	97	94	158	104	21
Средний	98	97	97	98	95	291	101	21
Большой	99	98	97	100	97	305	100	21
GENEALOGY_1								
Маленький	58	42	60	88	53	192	133	33
Средний	85	60	60	96	58	236	101	36
Большой	89	65	60	97	58	263	101	36
GENEALOGY_2								
Маленький	63	62	50	89	45	234	131	10
Средний	76	73	50	95	48	269	101	10
Большой	80	75	50	95	48	280	101	15

Таблица 11. Результаты измерения основных производных величин для класса COMPARABLE, оснащенного вручную (в процентах)

Набор тестов	B1	B2	B3	B4	B5	B6	B7	B8
COMPARABLE								
Маленький	56	30	48	30	14	124	103	30
Средний	98	73	48	100	48	152	101	30
Большой	100	82	48	100	48	148	101	30

Таблица 12. Средние значения производных величин, не зависящих от тестового набора (в процентах)

Код	**Описание**	**Значение**
B3	Процент выразимых предложений среди всех предложений разработчика	69
B9	Процент предложений, содержащих функции с одним аргументом, среди всех невыразимых предложений	62
B10	Процент предложений, содержащих импликации, среди всех невыразимых предложений.	20
B11	Процент предложений особого вида среди всех выведенных предложений.	12
B12	Процент релевантных предложений особого вида среди всех релевантных выведенных предложений.	7

Таблица 13. Средние значения производных величин, зависящих от тестового набора (в процентах)

Код	**Описание**	**Тестовый набор**		
		Маленький	Средний	Большой
B1	Точность	59	85	88
B2	Релевантность	56	77	78
B4	Выразимая полнота	73	93	93
B5	Абсолютная полнота	55	65	64
B6	Коэффициент усиления УР	299	406	408
B7	Коэффициент усиления ВУ	197	101	101
B8	Процент точек программы, где нет УР, но есть ВУ	13	16	18

Таблица 14. Средние значения производных величин, зависящих от тестового набора, для инвариантов циклов (в процентах)

Код	**Описание**	**Тестовый набор**		
		Маленький	Средний	Большой
B1	Точность	77	84	91
B2	Релевантность	68	78	81
B4	Выразимая полнота	100	100	100
B5	Абсолютная полнота	89	89	89
B6	Коэффициент усиления УР	1739	1961	2028
B7	Коэффициент усиления ВУ	103	100	100
B8	Процент точек программы, где нет УР, но есть ВУ	48	52	52

35 Результаты эксперимента различаются для библиотечных классов и классов из студенческих проектов. Это не отражено в таблицах, но среднее значение абсолютной полноты выведенных утверждений на большом тестовом наборе для первой группы классов равно 58%, а для второй — 75%. Это связано с более высоким качеством контрактов разработчика в повторно используемых библиотеках.

36 Большинство утверждений разработчика, невыразимых в грамматике *Daikon*, содержат функции с одним аргументом или импликации, в то время как большинство выведенных утверждений (особенно релевантных) имеют одну из трех форм, перечисленных на стр. 60. Показатели: $B9 \gg 0$; $B10 \gg 0$; $B11 \ll 100$; $B12 \ll 100$.

Таким образом, все теоретические предположения получили экспериментальное подтверждение. Кроме основных заключений сформулируем также ряд *наблюдений*, которые были сделаны автором в процессе эксперимента.

- Детектор *Daikon* отлично справляется с выводом простых, но крайне часто встречающихся утверждений, таких как `an_argument /= Void` в предусловиях или `some_attribute = an_argument` в постусловиях setter-

процедур. Такие утверждения носят технический характер, и разработчики не испытывают большого удовольствия от их постоянного повторения. Динамический вывод утверждений можно эффективно использовать для избавления разработчиков от этого бремени. В то же время, в области создания утверждений на высоком уровне абстракции приоритет пока остается за человеком (однако эту ситуацию может изменить подключение к процессу развертывания функций с аргументами).

- *Daikon* часто выводит «теоремы» — предложения, которые являются следствием других. Если это следование тривиально, то в такой избыточности нет ничего хорошего. Однако если следование можно обнаружить только зная семантику класса, наличие такой теоремы помогает лучше понять его спецификацию. Отметим, что разработчики создают такие теоремы крайне редко.

- Причиной вывода множества нерелевантных утверждений было использование шаблонов утверждений с участием битовых операций над целыми числами и других «слишком сложных» функций (в конце концов, их пришлось отключить). Много неверных утверждений порождает шаблон «неравенство переменных» (от него в результате тоже избавились). Множество неинтересных утверждений порождается сравнениями между несравнимыми переменными. Эту ситуацию предположительно можно исправить, реализовав анализ сравнимости (разд. 3.3).

  Кроме того, большое число неинтересных утверждений констатируют свойства, не специфичные для данной точки программы, а имеющие место в более общем случае. Это особенно характерно для инвариантов цикла, которые должны включать лишь постоянные свойства тех переменных, которые изменяются в цикле, а вместо этого включают свойства объемлющей подпрограммы и класса. Один из эвристических подходов к решению этой проблемы: отфильтровывать из выведенных инвариантов циклов те предложения, которые не содержат переменных, упоминаемых в теле цикла. По тем же соображениям в постусловиях функций можно отфильтровывать предложения, не содержащие переменной `Result`.

- Большой вред релевантности выведенных утверждений наносит несовершенство их представления на выходе детектора *Daikon*. Если в некоторой точке программы *Daikon* обнаруживает множество переменных, которые равны между собой, из них выбирается переменная-*лидер*. Утверждения, справедливые для всего множества, сообщаются только для лидера, и кроме того сообщается, что все остальные переменные из множества равны лидеру. Эта стратегия, обычно вполне пригодная, не подходит для случая, крайне часто встречающегося в классах *Eiffel*. Речь идет о ситуации, когда все переменные множества являются логическими или небольшими целочисленными константами. Вместо того, чтобы вывести равенство каждой из переменных константе, *Daikon* выбирает лидера и сообщает, что все остальные переменные равны ему. В результате утверждения становятся трудно читаемыми и неинтересными: ведь в действительности переменные между собой никак не связаны, их значения совпадают просто потому, что вероятность такого совпадения в случае логических констант равна 50%. Эта проблема сильно затруднила проведение эксперимента, так как оценка релевантности утверждений в этом случае очень неоднозначна.

- Для снижения доли неинтересных утверждений можно попробовать усовершенствовать механизм подавления одних утверждений другими (разд. 2.5). Стандартное использование этого механизма: подавление предусловий и постусловий инвариантом класса (эта функция не используется в *CITADEL*, так как инвариант класса в *Eiffel* строится по нестандартным правилам). Можно определить и более сложные правила, по которым инвариант класса A будет подавлять утверждения о переменной $x : A$. Как было упомянуто выше, подавление может быть также использовано для перехода от плоских форм выводимых контрактов к иерархическим. Не так давно в системе *Daikon* появился новый формат файла объявлений точек программы, который позволяет более гибко работать с иерархией подавления.

- Как показал эксперимент, модульные тесты в действительности неплохо подходят для вывода утверждений. При этом качественное улучшение тестового производит гораздо лучший эффект на результаты вывода, чем количественное увеличение.

4.4. Экспериментальная проверка метода сведения предусловий

Предложенный в разд. 2.1 метод сведения предусловий для улучшения качества тестовых наборов и поиска программных ошибок был отдельно проверен автором на двух специально подготовленных примерах.

Первый пример — подпрограмма решения квадратного уравнения. Ее код приведен в листинге 1.

Листинг 1. "Ошибочная подпрограмма решения квадратного уравнения"

```
solve_quadratic_equation (a, b, c: DOUBLE) is
    -- Solve quadratic equation with coefficients ‘a’, ‘b’ and ‘c’,
    -- put number of solutions into ‘solution_count’ and solutions
       into ‘x1’ and ‘x2’
  local
    d: DOUBLE
  do
    d := discriminant (a, b, c)
    if not approx_less_equal (d, 0) then
      solution_count := 2
      x1 := (-b + sqrt (d)) / (2 * a)
      x2 := (-b - sqrt (d)) / (2 * a)
    elseif approx_equal (d, 0) then
      solution_count := 1
      x1 := -b / (2 * a)
    else
      solution_count := 0
    end
  ensure
    case_positive: not approx_less_equal (discriminant (a, b, c),
       0.0) implies solution_count = 2
    case_zero: approx_equal (discriminant (a, b, c), 0.0) implies
       solution_count = 1
    case_negative: approx_less (discriminant (a, b, c), 0.0)
       implies solution_count = 0
    case_one: solution_count = 1 implies approx_equal (a * x1^2 + b
       * x1 + c, 0.0)
    case_two: solution_count = 2 implies approx_equal (a * x1^2 + b
       * x1 + c, 0.0) and approx_equal (a * x2^2 + b * x2 + c, 0.0)
  end
```

При тестировании подпрограммы было случайным образом сгенерировано 1000 тестовых случаев, причем значения коэффициентов выбирались из диапазона $[-10; 10]$. На первой же итерации детектор утверждений в числе других вывел в предусловии подпрограммы предложение `a /= 0`. Снабдив следующий тестовый набор случаем, нарушающим это предложение, обнаруживаем ошибку деления на ноль.

Второй пример — функция, определяющая, является ли одна строка суффиксом другой. Код функции приведен в листинге 2.

Листинг 2. "Ошибочная подпрограмма поиска суффикса строки"

```
ends_with (str, substr: STRING): BOOLEAN is
    -- Does ‘str’ end with ‘substr’?
  require
    str_exists: str /= Void
    substr_exists: substr /= Void
  local
    i: INTEGER
  do
    Result := True
    from
      i := 1
    until
      i > substr.count
    loop
      if substr.item (i) /= str.item (str.count - substr.count + i)
         then
        Result := False
        i := substr.count
      end
      i := i + 1
    end
  ensure
    definition: Result = substr.same_string (str.substring
       (str.count - substr.count + 1, str.count))
  end
```

При тестировании этой подпрограммы оригинальный тестовый набор содержал две качественно различные группы тестовых случаев. В первой строка `str` генерировалась из букв и цифр случайным образом, а подстрока `substr` выбиралась как ее случайный суффикс (таким образом, результат функции

всегда оказывался положительным). Во второй группе обе строки генерировались из букв и цифр случайным образом, однако длина строки `substr` всегда выбиралась меньшей длины строки `str` (не очень естественно для случайного тестирования, однако при системном тестировании такая ситуация вполне возможна). Это свойство обнаруживалось на первой же итерации работы *Daikon*. Его преднамеренное нарушение (выбор длины строки `substr` большим, чем строки `str`) приводит к обнаружению ошибки выхода индекса за границы допустимого диапазона.

Заключение

В настоящей работе были решены следующие задачи.

1. Выявлены свойства и разработана техника динамического вывода утверждений в чистом объектно-ориентированном языке с поддержкой проектирования по контракту.

2. Реализовано инструментальное средство *CITADEL* — *Eiffel*-интерфейс к детектору утверждений *Daikon*.

3. Проведено исследование применимости динамического вывода утверждений в языке *Eiffel*.

В результате исследования было подтверждено, что динамический вывод утверждений в *Eiffel* может быть использован для усиления и исправления контрактов разработчика, а также для улучшения качества тестовых наборов. Однако автоматизированный вывод не способен на данный момент заменить творческую работу человека по созданию спецификаций.

Использованные в настоящей работе инструменты (*Daikon* и *CITADEL*) неидеальны, и есть множество возможностей для их совершенствования, которое приведет к росту как точности, так и полноты выводимых контрактов. Почти все эти возможности уже обсуждались в предыдущих главах, поэтому здесь лишь перечислим их.

- Использование функций с одним аргументом при развертывании переменных в *CITADEL*.

- Более широкое использовании условных утверждений в *Daikon*.

- Реализация полнофункционального наследования точек программы в *CITADEL*.

- Модификация формата представления утверждений о булевых и целочисленных константах в *Daikon*.

- Реализация анализа сравнимости переменных.
- Поддержка более сложных форм подавления одних точек программы другими.
- Использование автоматического случайного тестирования.

Немного подробнее о последнем пункте. В настоящее время на вход инструменту *CITADEL* подается целая программная система. Это означает, что если требуется обработать лишь один или несколько классов, и при этом достаточно полно протестировать их поведение, пользователю *CITADEL* приходится вручную создавать модульные тесты. Эта необходимость в значительной степени умаляет пользу от вывода утверждений. Поэтому одним из перспективных направлений является подключение к *CITADEL* системы автоматического тестирования (например, системы *AutoTest* [22]). Если это удастся, использование *CITADEL* превратится в *push-button inference* (по аналогии с *push-button testing* — тестированием по нажатию одной кнопки). Остается открытым вопрос автоматической генерации таких тестовых наборов, которые обеспечили бы высокое качество выводимых утверждений. Это одна из важнейших тем будущих исследований.

Кроме того, в будущем возможно проведение повторного исследования применимости динамического вывода утверждений в языке *Eiffel* после реализации всех указанных выше усовершенствований в инструментах. При новом исследовании важно не забывать уроки, усвоенные во время эксперимента описанного в настоящей работе.

- Следует выделять множество выведенных предложений, которые являются следствием утверждением разработчика. Это поможет избежать использования аппроксимации, описанной в разд. 4.2, и более честно вычислить коэффициент усиления.
- Следует ввести более формальные правила для определения неинтересных предложений и, по возможности, ввести какие-нибудь правила для определения чрезмерной спецификации. Это позволит относить к релевантным только те выведенные утверждения, которые действительно будут интересны разработчику.

- Следует оценивать предложения, выведенные для предусловий отдельно, так как усиление предусловий, в отличие от других видов утверждений, не является усилением контракта. Возможно, динамический вывод предусловий следует использовать исключительно для целей улучшения качества тестовых наборов.

Благодарности

Автор выражает благодарность профессору *Бертрану Мейеру*, который предложил тему настоящей работы и организовал сотрудничество с Высшей Политехнической Школой в Цюрихе. Автор благодарит коллегу из Высшей Политехнической Школы *Илинку Чиупу* за непосредственное участие в эксперименте и плодотворное сотрудничество на протяжении всех этапов работы над диссертацией. Кроме того, автор благодарит *Владимира Точилина* за помощь в проведении эксперимента и решении множества теоретических и практических проблем.

Список литературы

1. *Мейер Б.* Объектно-ориентированное конструирование программных систем. М.: Русская редакция, 2005.
2. *Davies J., Woodcock J.* Using Z: Specification, Refinement and Proof. Prentice Hall, 1996.
3. *Abrial J. R.* The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996.
4. *Guttag J. V., Horning J. J.* Larch: Languages and Tools for Formal Specification. New York: Springer, 1993.
5. *Сайт проекта JML.* http://www.eecs.ucf.edu/ leavens/JML
6. *Meyer B.* Design by Contract, Technical Report TR-EI-12/CO. Interactive Software Engineering Inc., 1986.
7. *Standard ECMA-367.* Eiffel: Analysis, Design and Programming Language. ECMA International, 2006. http://www.ecma-international.org/publications/standards/Ecma-367.htm
8. *Erst M. D.* Dynamically Discovering Likely Invariants. PhD Dissertation. University of Washington, 2000.
9. *Сайт проекта* Daikon. http://groups.csail.mit.edu/pag/daikon
10. *Ernst M. D., Perkins J. H., Guo P. J. et al.* The Daikon system for dynamic detection of likely invariants // Science of Computer Programming, vol. 69, n. 1–3, 2007.
11. *Daikon Invariant Detector User Manual*, 2007.
12. *Daikon Invariant Detector Developer Manual*, 2007.
13. *Perkins J. H., Ernst M. D.* Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants / ACM SIGSOFT FSE, 2004.

14. *Pacheco C., Ernst M. D.* Eclat: Automatic Generation and Classification of Test Inputs / ECOOP, 2005.

15. *Harder M., Mellen J., Ernst M. D.* Improving test suites via operational abstraction / ICSE, 2003.

16. *Gupta N.* Generating test data for dynamically discovering likely program invariants / WODA, 2003.

17. *Gupta N., Heidepriem Z. V.* A new structural coverage criterion for dynamic detection of program invariants / 18th Annual International Conference on Automated Software Engineering, 2003.

18. *Nimmer J. W., Ernst M. D.* Automatic Generation of Program Specifications / International Symposium on Software Testing and Analysis, 2002.

19. *Kuzmina N., Gamboa R.* Dynamic Constraint Detection for Polymorphic Behavior / OOPSLA, 2006.

20. *Csallner C., Smaragdakis Y.* Dynamically Discovering Likely Interface Specifications / ICSE, 2006.

21. *Сайт проекта Gobo.* http://www.gobosoft.com/

22. *Инструмент AutoTest в проекте EiffelZone* http://eiffelzone.com/esd/tstudio

Приложение 1.

Некоторые аннотированные классы

В приложении приведены листинги нескольких классов, аннотированных при помощи *CITADEL*. Выведенные предложения можно отличить от оригинальных контрактов по метке `inferred`.

Листинг 3. Аннотированная версия класса BASIC_ROUTINES

```
class BASIC_ROUTINES

feature
  -- Conversion
  charconv (i: INTEGER): CHARACTER is
      -- Character associated with integer value ‘i’
    do
      Result := (i & 0x000000FF).to_character_8
    ensure then
      inferred: Result >= 0
    end

feature
  -- Basic operations
  abs (n: INTEGER): INTEGER is
      -- Absolute value of ‘n’
    do
      if n < 0 then
        Result := - n
      else
        Result := n
      end
    ensure
      inferred: Result >= 0
      inferred: n = 0 implies Result = 0
      inferred: Result = 0 implies n = 0
      inferred: Result >= n
      non_negative_result: Result >= 0
    end
```

```
sign (n: INTEGER): INTEGER is
    -- Sign of ‘n’:
    -- -1 if ‘n’ < 0
    --  0 if ‘n’ = 0
    -- +1 if ‘n’ > 0
  do
    if n < 0 then
      Result := -1
    elseif n > 0 then
      Result := +1

    end
  ensure
    inferred: -- Result one of { -1, 0, 1 }
    inferred: n = 0 implies Result = 0
    inferred: Result = 0 implies n = 0
    correct_negative: (n < 0) = (Result = -1)
    correct_zero: (n = 0) = (Result = 0)
    correct_positive: (n > 0) = (Result = +1)
  end

rsign (r: REAL): INTEGER is
    -- Sign of ‘r’:
    -- -1 if ‘r’ < 0
    --  0 if ‘r’ = 0
    -- +1 if ‘r’ > 0
  do
    if r < 0 then
      Result := -1
    elseif r > 0 then
      Result := +1

    end
  ensure
    inferred: -- Result one of { -1, 0, 1 }
    correct_negative: (r < 0) = (Result = -1)
    correct_zero: (r = 0) = (Result = 0)
    correct_positive: (r > 0) = (Result = +1)
  end
```

```
  bottom_int_div (n1, n2: INTEGER): INTEGER is
      -- Greatest lower bound of the integer division of ‘n1’ by
         ‘n2’
    require else
      inferred: n2 /= 0
    do
      Result := n1 // n2
      if n1 >= 0 xor n2 > 0 then
        if (n1 \\ n2) /= 0 then
          Result := Result - 1
        end
      end
    ensure then
      inferred: n1 = 0 implies Result = 0
    end

  up_int_div (n1, n2: INTEGER): INTEGER is
      -- Least upper bound of the integer division
      -- of ‘n1’ by ‘n2’
    require else
      inferred: n2 /= 0
    do
      Result := n1 // n2
      if not (n1 >= 0 xor n2 > 0) then
        if (n1 \\ n2) /= 0 then
          Result := Result + 1
        end
      end
    ensure then
      inferred: n1 = 0 implies Result = 0
    end

end
```

Листинг 4. Аннотированная версия класса FRACTION_2

```
class FRACTION_2

inherit

  NUMERIC
    redefine
```

```
      out
    end

create

  make

feature
  -- Creation
  make (a, b: INTEGER) is
      -- Creation procedure.
    require
      inferred: b /= 0
      denominator_not_zero: b /= 0
    do
      n := a
      d := b
      reduce
    ensure then
      inferred: n = 0 implies a = 0
      inferred: a = 0 implies n = 0
      inferred: b \\ d = 0
    end

feature
  -- Access
  one: like Current is
      -- Neutral element for "*" and "/"
    do
      create Result.make (1, 1)
    ensure then
      inferred: n = old n
      inferred: d = old d
      inferred: gcd = Result.n
      inferred: gcd = Result.d
      inferred: gcd = Result.gcd
      inferred: gcd = old gcd
      inferred: Result /= Void
      inferred: Result.zero /= Void
      inferred: Result.prefix "+" /= Void
      inferred: Result.prefix "-" /= Void
```

```
      inferred: Result.out /= Void
      result_exists: Result /= Void
    end

  zero: like Current is
      -- Neutral element for "+" and "-"
    do
      create Result.make (0, 1)
    ensure then
      inferred: n = old n
      inferred: d = old d
      inferred: gcd = Result.d
      inferred: gcd = Result.gcd
      inferred: gcd = old gcd
      inferred: Result /= Void
      inferred: Result.one /= Void
      inferred: Result.prefix "+" /= Void
      inferred: Result.prefix "-" /= Void
      inferred: Result.n = 0
      inferred: Result.out /= Void
      result_exists: Result /= Void
    end

feature
  -- Status report
  divisible (other: like Current): BOOLEAN is
      -- May current object be divided by ‘other’?
    require else
      inferred: gcd = other.gcd
      inferred: other /= Void
      inferred: other.one /= Void
      inferred: other.zero /= Void
      inferred: other.prefix "+" /= Void
      inferred: other.prefix "-" /= Void
      inferred: other.d /= 0
      inferred: other.out /= Void
      other_exists: other /= Void
    do
      Result := (other.n /= 0)
    ensure then
      inferred: n = old n
```

```
    inferred: d = old d
    inferred: gcd = other.gcd
    inferred: gcd = old gcd
    inferred: gcd = old other.gcd
    inferred: other.n = old other.n
    inferred: other.d = old other.d
    inferred: (other.n /= 0) = (Result)
    inferred: (other.n = 0) = (not Result)
    inferred: other.n = 0 implies d \\ other.d = 0
    inferred: other.n = 0 implies gcd > other.n
    inferred: other.n = 0 implies gcd >= other.d
    inferred: other.n = 0 implies n \\ other.d = 0
    inferred: other.n = 0 implies -- other.d one of { -1, 1 }
    inferred: other.one /= Void
    inferred: other.zero /= Void
    inferred: other.prefix "+" /= Void
    inferred: other.prefix "-" /= Void
    inferred: other.d /= 0
    inferred: other.out /= Void
  end

exponentiable (other: NUMERIC): BOOLEAN is
    -- May current object be elevated to the power ‘other’?
  require else
    inferred: other /= Void
    inferred: other.one /= Void
    inferred: other.zero /= Void
    inferred: other.prefix "+" /= Void
    inferred: other.prefix "-" /= Void
    other_exists: other /= Void
  do
    Result := false
      -- Roots (that you usually will get with fractions in the
         power) are real and therefore no representation as
         (integer-)fraction is possible.
  ensure then
    inferred: n = old n
    inferred: d = old d
    inferred: gcd = old gcd
    inferred: not Result
  end
```

```
feature
  -- Basic operations
  infix "+" (other: like Current): like Current is
      -- Sum with ‘other’ (commutative).
    require else
      inferred: gcd = other.gcd
      inferred: other /= Void
      inferred: other.one /= Void
      inferred: other.zero /= Void
      inferred: other.prefix "+" /= Void
      inferred: other.prefix "-" /= Void
      inferred: other.d /= 0
      inferred: other.out /= Void
      other_exists: other /= Void
    do
      create Result.make (n * other.d + other.n * d, d * other.d)
      Result.reduce
    ensure then
      inferred: n = old n
      inferred: d = old d
      inferred: gcd = Result.gcd
      inferred: gcd = old gcd
      inferred: gcd = old other.gcd
      inferred: Result /= Void
      inferred: Result.one /= Void
      inferred: Result.zero /= Void
      inferred: Result.prefix "+" /= Void
      inferred: Result.prefix "-" /= Void
      inferred: Result.d /= 0
      inferred: Result.out /= Void
      inferred: Result.n = 0 implies n = 0
      inferred: Result.n = 0 implies old other.n = 0
      result_exists: Result /= Void
      commutative: equal (Result, other + Current)
    end

  infix "-" (other: like Current): like Current is
      -- Result of subtracting ‘other’
    require else
      inferred: gcd = other.gcd
```

```
    inferred: other /= Void
    inferred: other.one /= Void
    inferred: other.zero /= Void
    inferred: other.prefix "+" /= Void
    inferred: other.prefix "-" /= Void
    inferred: other.d /= 0
    inferred: other.out /= Void
    other_exists: other /= Void
  do
    create Result.make (n * other.d - other.n * d, d * other.d)
    Result.reduce
  ensure then
    inferred: n = old n
    inferred: d = old d
    inferred: gcd = Result.gcd
    inferred: gcd = old gcd
    inferred: gcd = old other.gcd
    inferred: Result /= Void
    inferred: Result.one /= Void
    inferred: Result.zero /= Void
    inferred: Result.prefix "+" /= Void
    inferred: Result.prefix "-" /= Void
    inferred: Result.d /= 0
    inferred: Result.out /= Void
    result_exists: Result /= Void
  end

infix "*" (other: like Current): like Current is
    -- Product by ‘other’
  require else
    inferred: gcd = other.gcd
    inferred: other /= Void
    inferred: other.one /= Void
    inferred: other.zero /= Void
    inferred: other.prefix "+" /= Void
    inferred: other.prefix "-" /= Void
    inferred: other.d /= 0
    inferred: other.out /= Void
    other_exists: other /= Void
  do
    create Result.make (n * other.n, d * other.d)
```

```
    Result.reduce
  ensure then
    inferred: n = old n
    inferred: d = old d
    inferred: gcd = Result.gcd
    inferred: gcd = old gcd
    inferred: gcd = old other.gcd
    inferred: Result /= Void
    inferred: Result.one /= Void
    inferred: Result.zero /= Void
    inferred: Result.prefix "+" /= Void
    inferred: Result.prefix "-" /= Void
    inferred: Result.d /= 0
    inferred: Result.out /= Void
    inferred: n = 0 implies Result.n = 0
    inferred: old other.n = 0 implies Result.n = 0
    result_exists: Result /= Void
  end

infix "/" (other: like Current): like Current is
    -- Division by ‘other’
  require else
    inferred: gcd = other.gcd
    inferred: other /= Void
    inferred: other.one /= Void
    inferred: other.zero /= Void
    inferred: other.prefix "+" /= Void
    inferred: other.prefix "-" /= Void
    inferred: other.n /= 0
    inferred: other.d /= 0
    inferred: other.out /= Void
    other_exists: other /= Void
    good_divisor: divisible (other)
  do
    create Result.make (n * other.d, d * other.n)
    Result.reduce
  ensure then
    inferred: n = old n
    inferred: d = old d
    inferred: gcd = Result.gcd
    inferred: gcd = old gcd
```

```
    inferred: gcd = old other.gcd
    inferred: Result /= Void
    inferred: Result.one /= Void
    inferred: Result.zero /= Void
    inferred: Result.prefix "+" /= Void
    inferred: Result.prefix "-" /= Void
    inferred: Result.d /= 0
    inferred: Result.out /= Void
    inferred: n = 0 implies Result.n = 0
    inferred: Result.n = 0 implies n = 0
    result_exists: Result /= Void
  end

prefix "+": like Current is
    -- Unary plus
  do
    create Result.make (n, d)
  ensure then
    inferred: n = Result.n
    inferred: n = old n
    inferred: d = Result.d
    inferred: d = old d
    inferred: gcd = Result.gcd
    inferred: gcd = old gcd
    inferred: Result /= Void
    inferred: Result.one /= Void
    inferred: Result.zero /= Void
    inferred: Result.prefix "-" /= Void
    inferred: Result.out /= Void
    result_exists: Result /= Void
  end

prefix "-": like Current is
    -- Unary minus
  do
    create Result.make (- n, d)
    Result.reduce
  ensure then
    inferred: n = old n
    inferred: d = old d
    inferred: gcd = Result.gcd
```

```
      inferred: gcd = old gcd
      inferred: Result /= Void
      inferred: Result.one /= Void
      inferred: Result.zero /= Void
      inferred: Result.prefix "+" /= Void
      inferred: Result.d /= 0
      inferred: Result.d >= -1
      inferred: Result.out /= Void
      inferred: n = 0 implies Result.n = 0
      inferred: Result.n = 0 implies n = 0
      inferred: d \\ Result.d = 0
      inferred: d <= Result.d
      inferred: Result.d \\ d = 0
      result_exists: Result /= Void
    end

feature
  -- Values
  n, d: INTEGER
      -- Numerator and denominator.

feature
  -- Reduction
  gcd: INTEGER is
      -- Calculate the greatest common divisor of d and n.
    local
      m, l: INTEGER
    do
      if d = 0 then
        Result := n.abs
      elseif n = 0 then
        Result := d.abs

      else
          -- If neither d nor n are equal to 0 calculate the gcd
        from
          m := d.abs
          l := n.abs
        invariant
          inferred: one /= Void
          inferred: zero /= Void
```

```
        inferred: prefix "+" /= Void
        inferred: prefix "-" /= Void
        inferred: n /= 0
        inferred: d /= 0
        inferred: out /= Void
        inferred: l >= 1
        inferred: m >= 1
        inferred: Result = 0
        inferred: l > Result
        inferred: m > Result
        m > 0
        l > 0
      variant
        m.max (l)
      until
        m = l
      loop
        if m > l then
          m := m - l
        else
          l := l - m
        end
      end
      Result := m
    end
  ensure then
    inferred: n = old n
    inferred: d = old d
    inferred: Result >= 1
    inferred: n \\ Result = 0
    inferred: d \\ Result = 0
  end

reduce is
    -- Reduce the fraction to the smallest possible numerator and
       denominator.
  require else
    inferred: gcd >= 1
    inferred: n \\ gcd = 0
    inferred: d \\ gcd = 0
  local
```

```
      div: INTEGER
    do
      div := gcd
      n := n // div
      d := d // div
      if n < 0 and d < 0 then
        n := n.abs
        d := d.abs
      end
    ensure then
      inferred: n = 0 implies old n = 0
      inferred: old n = 0 implies n = 0
      inferred: old d \\ d = 0
      inferred: gcd <= old gcd
    end

feature
  -- Output
  out: STRING is
      -- Display the fraction.
    do
      Result := (n.out + "/" + d.out)
    ensure then
      inferred: n = old n
      inferred: d = old d
      inferred: gcd = old gcd
      inferred: Result /= Void
      inferred: not Result.is_empty
      inferred: gcd < Result.count
      out_not_void: Result /= Void
    end

invariant

  inferred: one /= Void
  inferred: zero /= Void
  inferred: prefix "+" /= Void
  inferred: prefix "-" /= Void
  inferred: d /= 0
  inferred: gcd = 1
  inferred: out /= Void
```

```
  denominator_not_zero: d /= 0

end
```